UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN INGENIERÍA DE SISTEMAS Y SERVICIOS PARA LA SOCIEDAD DE LA INFORMACIÓN

TRABAJO DE INVESTIGACIÓN

CompactRIO: Advanced Data Acquisition Systems Integration in CODAC Core System

---

Author:
Álvaro Bustos Benayas

Tutor:
Mariano Ruíz González

Julio 2015

RESUMEN

Las herramientas de configuración basadas en lenguajes de alto nivel como LabVIEW permiten el desarrollo de sistemas de adquisición de datos basados en hardware reconfigurable FPGA muy complejos en un breve periodo de tiempo. La estandarización del ciclo de diseño hardware/software y la utilización de herramientas como EPICS facilita su integración con la plataforma de adquisición y control ITER CODAC CORE SYSTEM (CCS) basada en Linux. En este proyecto se propondrá una metodología que simplificará el ciclo completo de integración de plataformas novedosas, como cRIO, en las que el funcionamiento del hardware de adquisición puede ser modificado por el usuario para que éste se amolde a sus requisitos específicos.

El objetivo principal de este proyecto fin de master es realizar la integración de un sistema cRIO NI9159 y diferentes módulos de E/S analógica y digital en EPICS y en CODAC CORE SYSTEM (CCS). Este último consiste en un conjunto de herramientas software que simplifican la integración de los sistemas de instrumentación y control del experimento ITER. Para cumplir el objetivo se realizarán las siguientes tareas:

- Desarrollo de un sistema de adquisición de datos basado en FPGA con la plataforma hardware CompactRIO. En esta tarea se realizará la configuración del sistema y la implementación en LabVIEW para FPGA del hardware necesario para comunicarse con los módulos: NI9205, NI9264,NI9401.NI9477, NI9426, NI9425 y NI9476
- Implementación de un driver software utilizando la metodología de AsynDriver para integración del cRIO con EPICS. Esta tarea requiere definir todos los records necesarios que exige EPICS y crear las interfaces adecuadas que permitirán comunicarse con el hardware.
- Implementar la descripción del sistema cRIO y del driver EPICS en el sistema de descripción de plantas de ITER llamado SDD. Esto automatiza la creación de las aplicaciones de EPICS que se denominan IOCs.

SUMMARY

The configuration tools based in high-level programing languages like LabVIEW allows the development of high complex data acquisition systems based on reconfigurable hardware FPGA in a short time period. The standardization of the hardware/software design cycle and the use of tools like EPICS ease the integration with the data acquisition and control platform of ITER, the CODAC Core System based on Linux. In this project a methodology is proposed in order to simplify the full integration cycle of new platforms like CompactRIO (cRIO), in which the data acquisition functionality can be reconfigured by the user to fits its concrete requirements.

The main objective of this MSc final project is to develop the integration of a cRIO NI-9159 and its different analog and digital Input/Output modules with EPICS in a CCS. The CCS consists of a set of software tools that simplifies the integration of instrumentation and control systems in the International Thermonuclear Reactor (ITER) experiment. To achieve such goal the following tasks are carried out:

- Development of a DAQ system based on FPGA using the cRIO hardware platform. This task comprehends the configuration of the system and the implementation of the mandatory hardware to communicate to the I/O adapter modules NI9205, NI9264, NI9401, NI9477, NI9426, NI9425 y NI9476 using LabVIEW for FPGA.
- Implementation of a software driver using the asynDriver methodology to integrate such cRIO system with EPICS. This task requires the definition of the necessary EPICS records and the creation of the appropriate interfaces that allow the communication with the hardware.
- Develop the cRIO system's description and the EPICS driver in the ITER plant description tool named SDD. This development will automate the creation of EPICS applications, called IOCs.

Table of Contents

Table of Figures

Table of Tables

RESUMEN

Las herramientas de configuración basadas en lenguajes de alto nivel como LabVIEW permiten el desarrollo de sistemas de adquisición de datos basados en hardware reconfigurable FPGA muy complejos en un breve periodo de tiempo. La estandarización del ciclo de diseño hardware/software y la utilización de herramientas como EPICS facilita su integración con la plataforma de adquisición y control ITER CODAC CORE SYSTEM (CCS) basada en Linux. En este proyecto se propondrá una metodología que simplificará el ciclo completo de integración de plataformas novedosas, como cRIO, en las que el funcionamiento del hardware de adquisición puede ser modificado por el usuario para que éste se amolde a sus requisitos específicos.

El objetivo principal de este proyecto fin de master es realizar la integración de un sistema cRIO NI9159 y diferentes módulos de E/S analógica y digital en EPICS y en CODAC CORE SYSTEM (CCS). Este último consiste en un conjunto de herramientas software que simplifican la integración de los sistemas de instrumentación y control del experimento ITER. Para cumplir el objetivo se realizarán las siguientes tareas:

- Desarrollo de un sistema de adquisición de datos basado en FPGA con la plataforma hardware CompactRIO. En esta tarea se realizará la configuración del sistema y la implementación en LabVIEW para FPGA del hardware necesario para comunicarse con los módulos: NI9205, NI9264,NI9401.NI9477, NI9426, NI9425 y NI9476
- Implementación de un driver software utilizando la metodología de AsynDriver para integración del cRIO con EPICS. Esta tarea requiere definir todos los records necesarios que exige EPICS y crear las interfaces adecuadas que permitirán comunicarse con el hardware.
- Implementar la descripción del sistema cRIO y del driver EPICS en el sistema de descripción de plantas de ITER llamado SDD. Esto automatiza la creación de las aplicaciones de EPICS que se denominan IOCs.

SUMMARY

The configuration tools based in high-level programing languages like LabVIEW allows the development of high complex data acquisition systems based on reconfigurable hardware FPGA in a short time period. The standardization of the hardware/software design cycle and the use of tools like EPICS ease the integration with the data acquisition and control platform of ITER, the CODAC Core System based on Linux. In this project a methodology is proposed in order to simplify the full integration cycle of new platforms like CompactRIO (cRIO), in which the data acquisition functionality can be reconfigured by the user to fits its concrete requirements.

The main objective of this MSc final project is to develop the integration of a cRIO NI-9159 and its different analog and digital Input/Output modules with EPICS in a CCS. The CCS consists of a set of software tools that simplifies the integration of instrumentation and control systems in the International Thermonuclear Reactor (ITER) experiment. To achieve such goal the following tasks are carried out:

- Development of a DAQ system based on FPGA using the cRIO hardware platform. This task comprehends the configuration of the system and the implementation of the mandatory hardware to communicate to the I/O adapter modules NI9205, NI9264, NI9401, NI9477, NI9426, NI9425 y NI9476 using LabVIEW for FPGA.
- Implementation of a software driver using the asynDriver methodology to integrate such cRIO system with EPICS. This task requires the definition of the necessary EPICS records and the creation of the appropriate interfaces that allow the communication with the hardware.
- Develop the cRIO system's description and the EPICS driver in the ITER plant description tool named SDD. This development will automate the creation of EPICS applications, called IOCs.

1 DOCUMENT STRUCTURE

1.1 Context

Control, Data Access and Communication (CODAC) is the central control system responsible for operating the ITER device. CODAC interfaces to more than 30 ITER plant systems containing actuators, sensors and local Instrumentation and Control (I&C). For the machine protection, interlock system and safety (personnel and environment) systems, are explicitly decoupled from CODAC and act fully independently. Control System Division, in charge of aforementioned tasks, is also responsible for the central interlock system and central safety system.

CODAC Core System (CCS) is the operating system for ITER and is based on Experimental, Physics and Industrial Control System (EPICS) and Control System Studio (CSS). Users who contribute to the development of ITER I&C System, such as ITER Domestic Agencies or industries working for ITER through contracts, uses CODAC and dedicated software distribution.

The CODAC Core System is a software package that is distributed by CODAC Section of ITER Organization for the development of the Plant System I&C. It includes the software for Mini-CODAC, Plant System Host (PSH) and Plant Controllers Fast (PCF) and it provides the plant system I&C developers the environment required to develop and test the software satisfying the ITER requirements.

The operating system for Mini-CODAC, PSH and PCF is an officially supported version of Red Hat Enterprise Linux (RHEL). The EPICS base is included in the distribution and is required for Mini-CODAC, PSH and PCF. The EPICS framework is the base for the Fast Controllers and PSH, and the EPICS CA protocol for access to plant system I&C over the Plant Operation Network (PON).

The ITER project is broken down into plant systems, known as the Plant Breakdown Structure (PBS). The plants have specific functional requirements, each requirement has to be treated separately with its own I&C System called Plant System I&C. In order to facilitate integration and control, CODAC has a functional categorization named Control Breakdown Structure. Plant System I&Cs are grouped into hierarchical control groups. Each of these control groups can have a number of servers dedicated to runtime activities such as archiving, control group management and configuration management for the control group.

A fast controller is a control system component defined as a plant system controller used to implement control loops or data acquisition in Plant System Instrumentation & Control at a rate faster than 100 Hz. In the current design it is implemented using PXI Express, ATCA or uTCA based solutions, the first ones to carry the I/O boards and the last one proposed for diagnostics. The intended capabilities for a PCF are:

- Data acquisition with accurate time stamping
- Actuation with precise timing
- Real-time exchange of data with other systems locally closed control loops in hard real-time.

In the specific case of the PXIe, the chassis is separated from the CPU standard industrial computer and interconnected using a PCIe link. The fast controller is composed by an industrial PC with two separated Network Interface Cards (NIC) one for Data Archive Network (DAN) and Synchronous Databus Network (SDN) and the other for the Plant Operation Network (PON). All the hardware mentioned for the PCF is integrated in a single I/C Cubicle. The link to the PXIe chassis is with a PCIe-PXIe bridge.

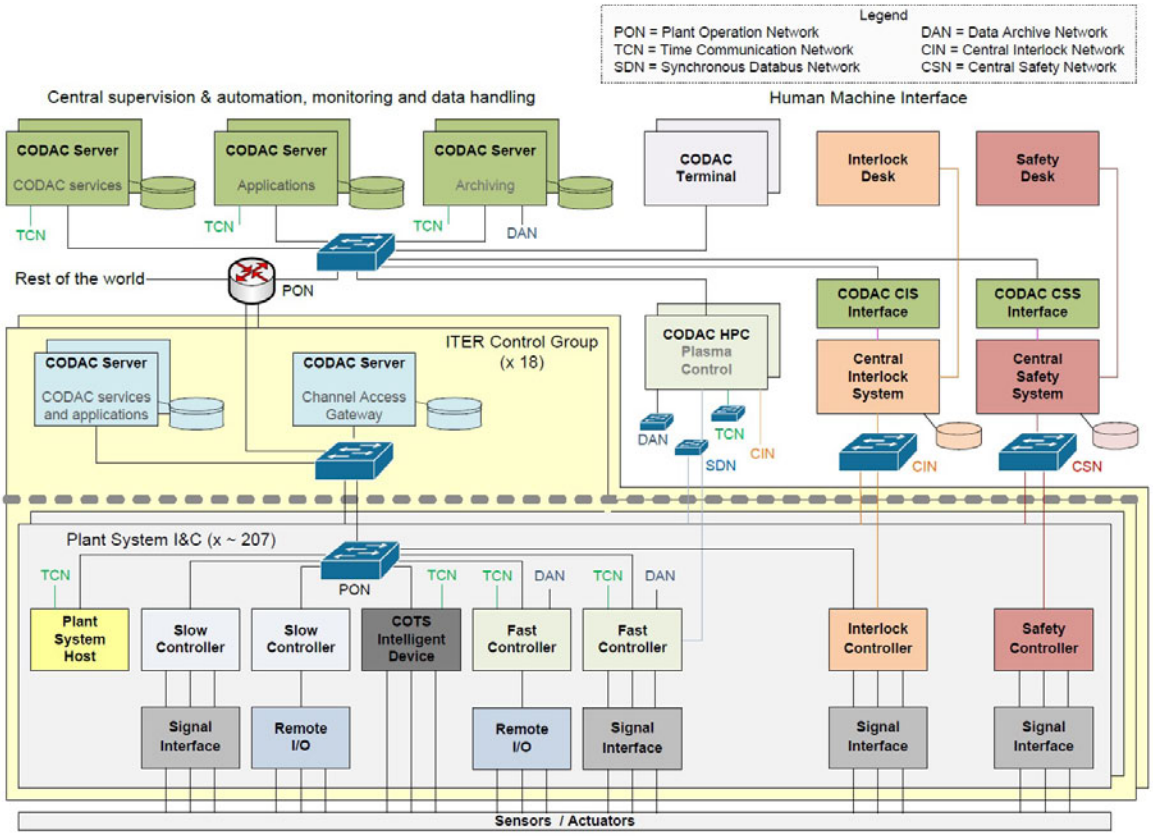

Fig. 1 Physical Architecture of ITER I&C Systems

1.2 Motivation

The ITER Fast Controller hardware catalogue [RD1] includes the use of FlexRIO and cRIO solutions for the implementation of control and data acquisition systems. cRIO is identified as the solution for I&C applications and FlexRIO for DAQ applications. The first uses a form factor defined by National Instruments and the later uses the PXI format standardized by PXI Systems Alliance [RD2]. Both technologies are implemented around the use of a Xilinx FPGA.

CODAC Core System [RD3] includes the Linux kernel modules and API libraries, NI-RIO Linux Device Driver, for using the FlexRIO and cRIO solutions. These platforms must be connected to the Fast Controller using a PCIe link available or in the MXI interfaces defined in FC hardware catalogue.

Not existing any standardization in ITER for the development of data acquisition and control systems with RIO devices, drives the necessity of a big amount of resources, understood as CODAC developers, to be deployed in order to accomplish any development using such RIO devices. Therefore a project to achieve such standardization has been launched by ITER Organization with the collaboration of the Instrumentation and Applied Acoustics Research Group of the Technical University of Madrid (UPM) [RD4].

IRIO project provides a set of design rules, templates and different software modules that allow CCS users to implement two different types of applications using cRIO and FlexRIO hardware [RD24]:

- EPICS IOCs using an EPICS device support.
- Standalone C/C++ applications.

The target applications for cRIO are basically the implementation of data acquisition and control systems in the FPGA. The cRIO FPGA can acquire digital and analog signals, process them in the FPGA and actuate accordingly to other analog or digital output signals. Of course, the information acquired, processed and generated in the FPGA can be monitored and controlled from an EPICS IOC using the IRIO EPICS device support. The target applications for FlexRIO are related with high sampling rate data acquisition and processing systems. FlexRIO devices can implement data acquisition, and process the data obtaining results that will be acquired by the EPICS device support. In general the acquisition and data movement at high sampling rate to EPICS is not considered because EPICS cannot manage such big amount of data.

1.3 IRIO Software Features

IRIO software library has been designed with the goal of simplifying the interface with RIO devices (compactRIO and FlexRIO). These devices are based on an FPGA which implies an infinite number of possible implementations depending on the FPGA project developer requirements. This means that the IRIO library user must know about how the RIO device has been configured, in order to implement correctly the software application. The user, will receive the FPGA *bitfile*, the file to be used by IRIO library to program the FPGA (RIO device), and the FPGA *header* file that contains the address mapping of the FPGA resources. Both files correspond to a specific RIO configuration.

Below it will be explained how the device is configured and how the user will be able to know how to use the library according to it.

The first thing that the user should know is in which software layer, related to the hardware, is located this library (see Fig. 2).

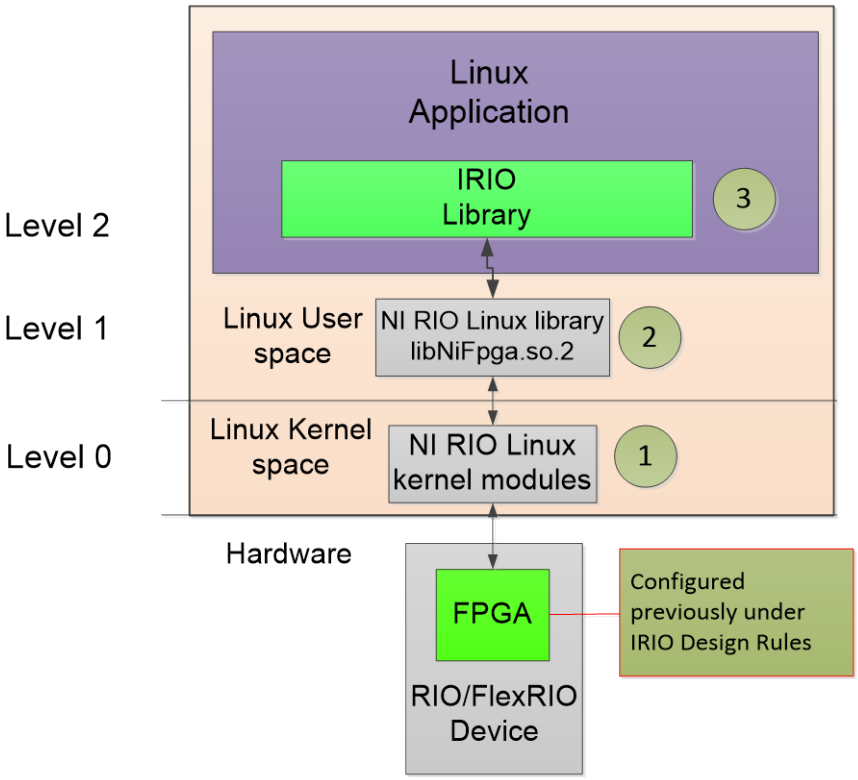

Fig. 2 Software layers used by IRIO Library

As every Linux device driver implementation, NI RIO Linux device driver (Level 0) [RD27], requires its corresponding user space library (Level 1) that permits the access to the RIO device. Finally IRIO library (Level 2) uses the NI RIO library to interface the user application with the hardware.

The main features of IRIO software are:

- Identify the resources and profiles implemented in the FLEXRIO/cRIO FPGA. These resources are accessible using the IRIO API that supports the interface with the FPGA, the implementation of Analog input data acquisition (using DMA), Signal Generation (DDS), digital I/O, image acquisition using CameraLink and serial line for CameraLink cameras configuration.
- Provide an EPICS device support implemented with asynDriver named NI-IRIO EPICS Device Driver including interfaces for asynOctet, asynint32, asynUInt32Digital, asynInt32Array, asynFloat64, asynFloat32Array, asynFloat64array. The EPICS device support contains a list of predefined reason/interfaces to map EPICS records to FPGA resources. This library is already implemented and is licensed with GPLv2.
- Include a library with C++ classes, named IRIOnds, for the implementation of EPICS device support using Nominal Device Support approach. The classes provided allow the

implementation of analog input, digital I/O, image acquisition and the use of the internal resource of the FPGA and RIO products.

1.4 Development Steps

This document summarizes the part of the corresponding IRIO Project related to cRIO technology which I have been developing as a member of the work team. Two sections can be distinguished in the document, the first section explains the methodology for building cRIO LabVIEW/FPGA applications, and the second explains how the NIRIO EPICS device support has been implemented.

The basic stages to integrate the EPICS and asynDriver support for National Instruments CompactRIO data acquisition and control devices connected to adapter modules with analog and digital input/output elements comprise the following items:

- CompactRIO technology characteristics and functionalities analysis and the LabVIEW/FPGA mechanisms offered to its configuration.
- Definition of a basic model of architecture that permits the management of the specific input/output resources
- Definition of a basic representation model for this technology in EPICS defining its records and Process Variables (PVs)
- Implementation of an asynDriver device driver supporting the defined PVs
- Creation of an EPICS Input Output Controller (IOC), which uses the implemented driver, to test its functionality through EPICS utilities
- Description of the solution using the tools provided by the CCS SDD-editor

1.5 Assumptions

It is assumed that the CODAC Core System, with the CompactRIO hardware installed, is the target operating system.

The software is implemented considering the features provided by NI-RIO Linux Device Driver, EPICS and asynDriver. The NI-RIO Linux Device Driver and IRIO Software Library are available in CCS to use this EPICS device support. Additionally the user needs to use a hardware implementation for the FPGA developed using the provided design rules.

It is assumed that the user is familiar with the CODAC Core System and has some basic knowledge about EPICS [RD6].

The user of this software is mainly the developer of applications for ITER Fast Controllers. These developers are going to develop applications using RIO hardware defined in ITER FC Hardware catalogue of three different categories:

- EPICS IOCs using the NI-RIO EPICS Device Support, including the use of SDD templates for RIO hardware.
- Specific EPICS device support using the NDS C++ classes for RIO hardware
- Standalone C/C++ applications using the RIO hardware.

1.6 Material Required

Material Required	
Hardware	National Instruments CompactRIO chassis NI-9159
	National Instruments CompactRIO adapter modules • NI-9425 • NI-9426 • NI-9476 • NI-9477 • NI-9205 • NI-9264 • NI-9401
	PCI Express Expansion Link NI-PXI-PCIe8362
	Personal Computer
	Industrial ECRIN system Computer (ITER Cubicle)
Software	Windows 7 SP1
	National Instruments LabVIEW 2013 SP1 f2 • FPGA Module 13.0.1 • CompactRIO 13.0 Support
	Linux RedHat 6.5 with CODAC Core System software distribution

1.7 Acronyms

AI	Analog Input
AO	Analog Output
ASIC	Application Specific Integrated Circuit
CLB	Configurable Logic Blocks
CODAC	Control, Data Access and Communication
CCS	CODAC core system
cRIO	Compact RIO
DI	Digital Input
DMA	Direct memory Access
DO	Digital Output

EPICS	Experimental Physics and Industrial Control System
FPGA	Field Programmable Gate Array
FlexRIO	Flexible RIO
HDD	Hard Disk Drive
HDL	Hardware Description Language
HMI	Human-Machine Interface
I/O	Input/Output
IO	ITER Organization
IOC	Input / Output Controller
IRIO	Intelligent RIO
I&C	Instrumentation and Control
LUT	Look-Up Tables
NI	National Instruments
OTP	OneTime Programmable
PBS	Plant Breakdown Structure
PC	Personal computer
PCF	Plant Controller Fast
PON	Plant Operation Network
PSH	Plant System Host
PV	Process Variable
RAM	Random Access Memory
RHEL	Red Hat Enterprise Linux
RIO	Reconfigurable Input/Output
SDD	Self-Description Data
SRAM	Static RAM
TCP	Transmission Control Protocol
VI	Virtual Instrument
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
1oo2	1 out of 2
2oo3 SD	2 out of 3 majority voting double decker system with diagnostics
ACQ	Acquisition
A/D	Analog to Digital
AI	Analog Input
AO	Analog Output
CIS	Central Interlock System
CLI	Command Line Interface
Comm	Communication
COTS	Commercial Off-the-Shelf
CRC	Cyclic Redundancy Check
cRIO	Compact Reconfigurable Input Output
CSCI	Computer Software Configuration Item
D/A	Digital to Analog
DC	Diagnostic Coverage
DI	Digital Input
DIAG	Diagnostic

DO	Digital Output
E	Enable
FPGA	Field Programmable Gate Array
FPIS	Fast PIS
GUI	Graphical User Interface
HWCI	Hardware Configuration Item
HMI	Human Machine Interface
ICS	Interlock Control System
IDM	ITER Document Management System
MOSI	Master Output Slave Input
MISO	Master Input Slave Output
OUT	Output of a block
PIS	Plant Interlock System
PTC	Proof Test Chassis
PTS	Proof Test System
PTT	Proof Test Tool
PTRN	Pattern. Data with a random or predefined behaviour
RIO	Reconfigurable Input/Output
RSE	Referenced Single Ended
SPI	Serial Peripheral Interface
UUT	Unit Under Test
VI	Virtual Instrument
CPU	Central Processing Unit
CODAC	COntrol, Data Access and Communication
DAQ	Data AcQuisition
FlexRIO	Flexible Reconfigurable Input/Output
FPGA	Field Programmable Gate Array
FPSC	Fast Plant System Controller
I&C	Instrumentation and Control
I/O	Input and Output
NI	National Instrument
PCDH	Plant Control Design Handbook
PCI	Peripheral Component Interconnect – computer bus standard
PCI Express	Peripheral Component Interconnect Express
PICMG	PCI Industrial Computer Manufacturers Group
PXI	PCI Extensions for Instrumentation
PXI Express	An evolution of PXI using PCI Express technologies
OS	Operating system
RIO	Reconfigurable input/output

2 COMPACTRIO PLATFORM OVERVIEW

2.1 Brief FPGA Basics

Field Programmable Gate Arrays (FPGAs) are programmable semiconductor devices that are based around a matrix of Configurable Logic Blocks (CLBs) connected through programmable interconnections. As opposed to Application Specific Integrated Circuits (ASICs), where the device is custom built for the particular design, FPGAs can be programmed to the desired application or functionality requirements. Although OneTime Programmable (OTP) FPGAs are available, the common types of FPGAs are SRAM-based where the modelled hardware hosted can be changed when the design evolves. Fig. 3 depicts the main elements which the FPGA is composed by.

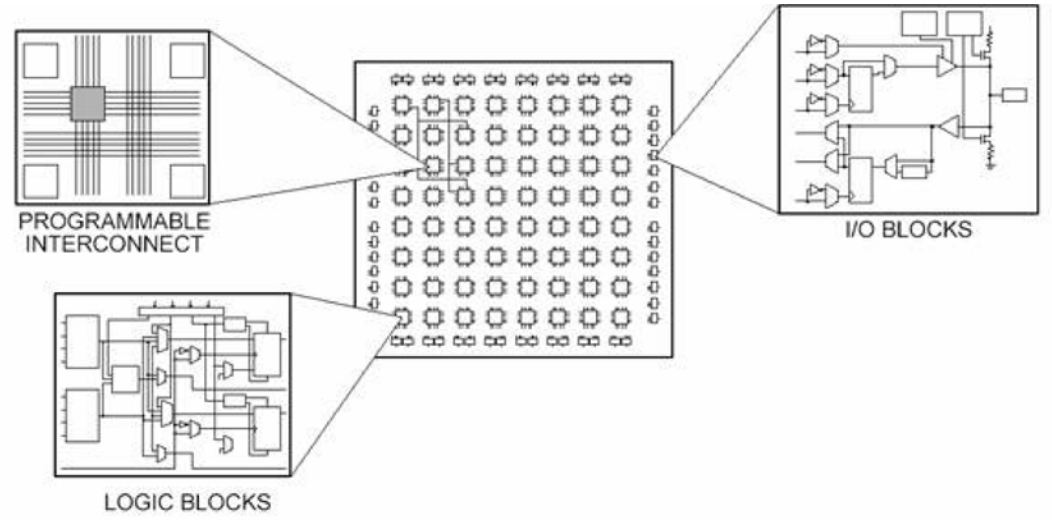

Fig. 3 Schema of the elements in a FPGA

The configurable logic blocks (CLBs), slices or logic cells, -depicted in Fig. 4- are the basic logic units of a FPGA. They are made up of: a configurable switch matrix with 4 or 6 inputs; some selection circuitry, like multiplexers; and flip-flops. Various FPGA families differ in the way flip-flops and LUTs are packaged together. The switch matrix is highly flexible and can be configured to handle combinatorial logic, shift registers or RAM.

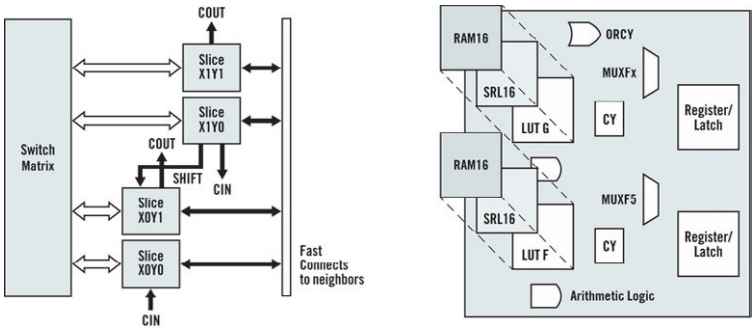

Fig. 4 Configurable Logic Block Structural Scheme

The flexible interconnection of the FPGA routes the signals between CLBs and I/Os. There are different types of routing, from the interconnection between CLBs to fast horizontal and vertical lines crossing the device to global low-skew routing for clocking and other global signals. The design software makes the interconnection routing task hidden to the user, unless necessity, significantly reducing design complexity. I/Os in FPGAs are grouped in banks with each bank independently able to support different I/O standards. Today's FPGAs provide over a dozen I/O banks, thus allowing flexibility in I/O support. Embedded Block RAM memory is available in most FPGAs, which allows for on-chip memory in your design. Digital clock management is provided by most FPGAs in the industry and also phase-looped locking that provide precision clock synthesis combined with jitter reduction and filtering.

Memory resources are another key specification to consider when selecting FPGAs. Depending on the FPGA family the on-board RAM can be configured in different block sizes. Digital signal processing algorithms often need to keep track of an entire block of data, or the coefficients of a complex equation, and without on-board memory, many processing functions do not fit within the configurable logic of a FPGA chip.

2.2 FPGA Design Tools

The method to build the logic that will be placed in the FPGA is modelling the behaviour of the system using development tools and then compile them down to a configuration file or bitstream that contains information on how the components should be wired together.

Hardware description languages (HDLs) such as VHDL and Verilog are textual languages for architecting a circuit. The syntax requires signals to be mapped or connected from external I/O ports to internal signals, which ultimately are wired to the modelled hardware entities. However, the modelled hardware behaviour is hard to be visualized in a sequential line-by-line flow textual language. For the verification of the logic created is a common practice to write test benches in HDL to wrap around and exercise the FPGA design by asserting inputs and verifying outputs. The test bench and FPGA code run in a simulation environment that models the hardware timing behaviour of the FPGA chip and displays the input and output signals to the designer for test validation. The process of creating the HDL test bench and executing the simulation requires at least four times more than creating the original FPGA HDL design itself. Once the text-based model of the hardware is verified through several steps, synthesizes the HDL down into a configuration file or bitstream that contains information on how the components should be wired together. As part of this multi-step process, mapping of signal names to the pins on the FPGA chip have to be done.

The rise of high-level synthesis (HLS) design tools, such as NI LabVIEW system design software, changes the rules of FPGA modelling and delivers new technologies that convert graphical block diagrams into digital hardware circuitry. The LabVIEW programming environment is suited for FPGA modelling being easier for the designer to recognize parallelism and data flow. Also VHDL can be integrated into LabVIEW FPGA designs. To simulate and verify the behaviour of the FPGA logic, LabVIEW offers features directly in the development environment. LabVIEW FPGA compilation tools automate the compilation process, highlighting errors, if occur, and critical paths if timing errors appears in the design.

2.3 RIO Platform Architecture

The reconfigurable I/O (RIO) architecture combines the graphical programming environment with a reconfigurable FPGA and I /O Modules for measurement and/or acquisition, see Fig. 5. FPGAs are used for creating highly customizable and reconfigurable platforms implementing processing and control tasks with hardware circuitry and the capacity to perform multiparallel operations within a single clock cycle. Orchestrate with a processor offloaded by the FPGA and used to configure the FPGA, interface with other peripherals, log data, run applications, etc. and I /O Modules directly connected to the FPGA for interfacing with other devices.

The reconfigurable FPGA is the core of the RIO hardware system architecture; it is directly connected to the I/O modules for high-performance access to the I/O circuitry of each module and high timing, triggering, and synchronization flexibility. In some of the RIO devices, like FlexRIO, each module is connected directly to the FPGA rather than through a bus, there is almost no control latency for system response compared to other industrial controllers.

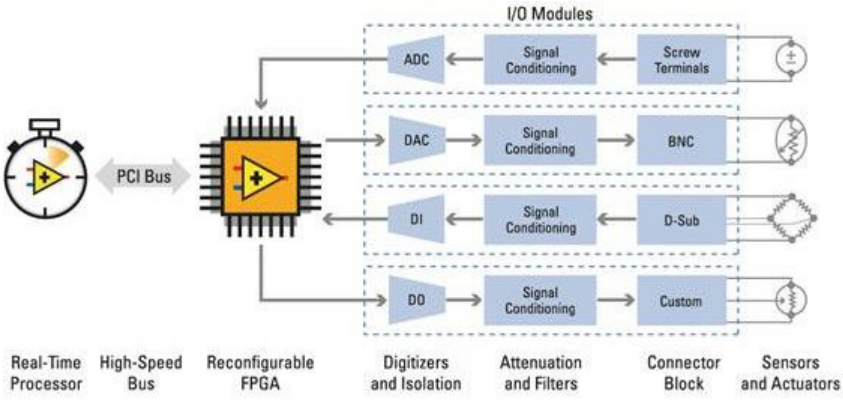

Fig. 5 NI RIO Architecture Diagram

2.3.1 RIO for Compact Embedded Applications

CompactRIO is a small, rugged RIO system for embedded and prototyping applications, configurable with four- eight- and fourteen- slot backplanes. It contains two main components: a reconfigurable FPGA in a chassis, and the interchangeable industrial I/O modules.

NI MXI-Express RIO is an extension of the National Instruments RIO platform, MXI-Express RIO consist of an eight- or fourteen-slot backplane and a FPGA in each CompactRIO chassis.

MXI-Express RIO combines a customizable field-programmable gate array (FPGA) and modular I/Os.

MXI-Express RIO differs from the other devices in the NI RIO platform in that it decouples the FPGA and the modular I/Os from the processor and allows multiple chassis to communicate to the same controller over x1 cabled PCI Express interface with 250 MB/s theoretical bandwidth from up to six daisy chained chassis. In fact its I/O modularity is one of the reasons of CompactRIO for not being as fast as other elements of the RIO family like FlexRIO, which drives the system to stablish an internal bus connection from the FPGA side to the I/O module side.

The cRIO chassis and I/O modules available in ITER hardware catalogue are listed in Table 1.

Table 1 ITER HW Catalogue for cRIO

Number Part	Description
NI-9159	MXI-Express 14 slot cRIO chassis
NI-9425	Sinking Digital Input Module. 7 µs, 32 channels 12 / 24 V 32 channels
NI-9426	Sourcing Digital Input Module. 7 µs, 32 channels 12 / 24 V 32 channels
NI-9476	Sourcing Digital Output Module. 500µs / 6-36V / 250 mA 32 channels
NI-9477	Sinking Digital Output Module 8 µs / 5-60V / 625 mA 32 channels
NI-9205	Analog Input Module 16-bit, ±200mV to ±10V, 250 kS/s 32 channels
NI-9264	Analog Output Module

Number Part	Description
	16-bit, ±10V, 25kS/s 16 channels
NI-9401	5V/TTL Bidirectional Digital I/O 8 channels

3 METHODOLOGY & DESIGN RULES

3.1 Introduction

This section explains the different steps that a developer must follow to build data acquisition applications using National Instruments RIO devices such as CompactRIO. These steps are part of the methodology proposed in the IRIO Project in order to integrate such devices implementing DAQ applications in CCS.

1. Selection of the RIO system. In this case, cRIO system.
2. Create a LabVIEW project. This section provides more details about this and LabVIEW project templates to simplify the development.
3. Write a VI using the specific rules and recommendations described in this document. Section 3.3 describes the rules to be used when creating a VI for cRIO.
4. Compile the VI and obtain the bitfile using LabVIEW for FPGA tools. This tool calls directly to the XILINX compiler simplifying the process to obtain the bitfile. Once you complete successfully this step you can test and debug your LabVIEW for FPGA application in the windows computer.
5. Before starting the test in a Linux machine, generate the header file of the design using the "FPGA Interface C API Generator" software application. Section 3.5 provides more details about this step.
6. Move (copy) the header file and bit file obtained to the Linux machine and start testing your applications in the RedHat environment.

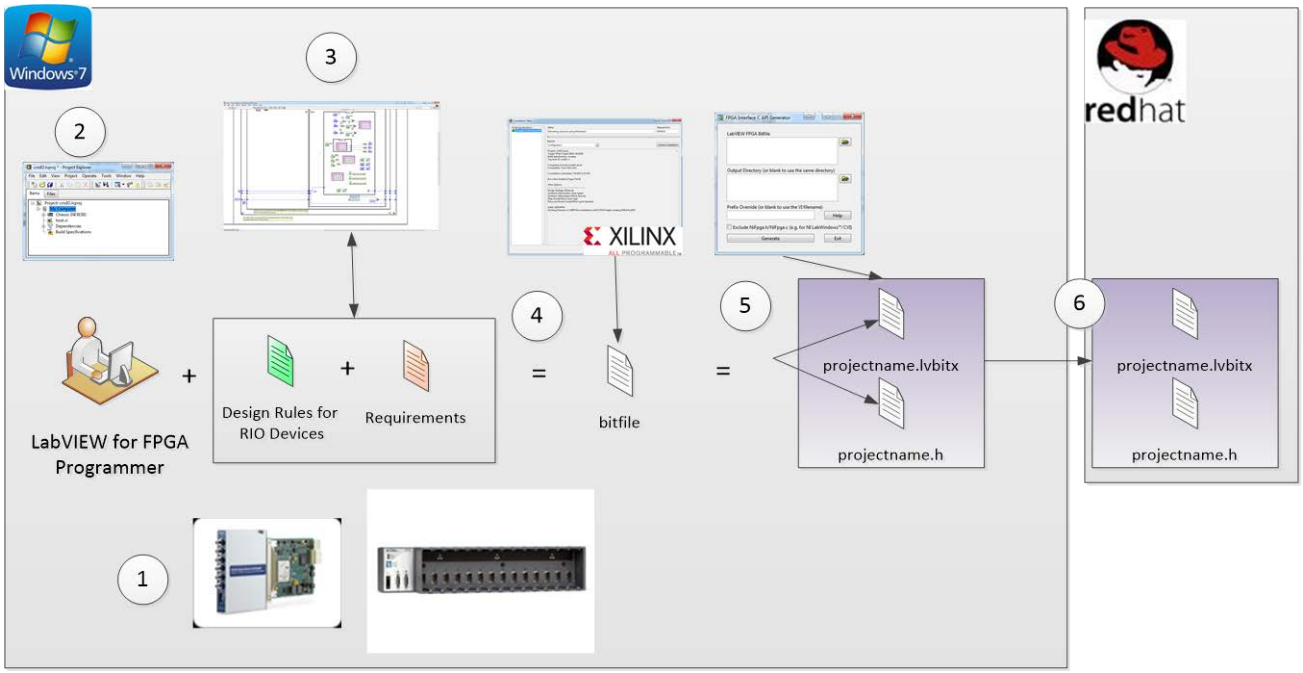

Fig. 6 Design Flow for RIO devices

3.2 LabVIEW Project Structure for a RIO Device

The first step for using a RIO device is to develop a project in LabVIEW for FPGA. This software only runs on Windows OS and provides all of the tools to configure/program the FPGAs. The configuration is made using a bitfile generated by the software environment.

The LabVIEW project has to contain the FPGA target, in this case the NI9159 cRIO platform, and the cRIO I/O adapter modules allocated in it. Fig. 7 depicts the LabVIEW menu to include a RIO device in the project.

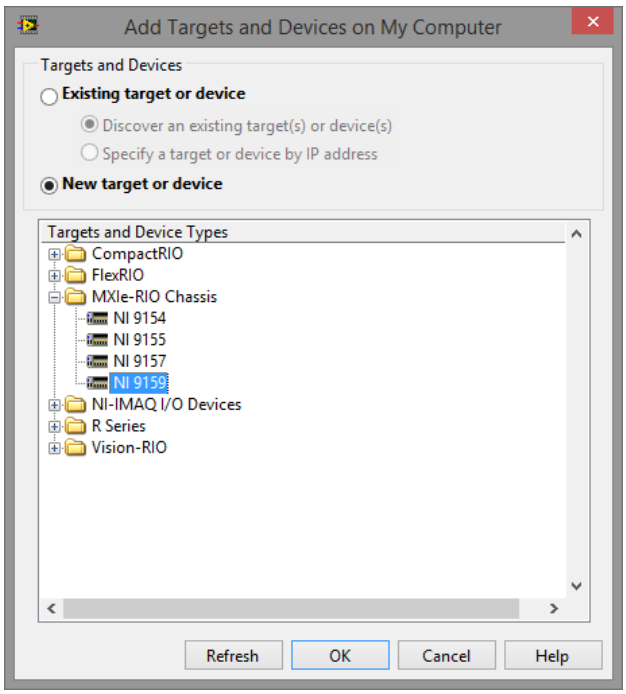

Fig. 7 Adding MXIe-RIO Chassis NI-9159 in LabVIEW Project

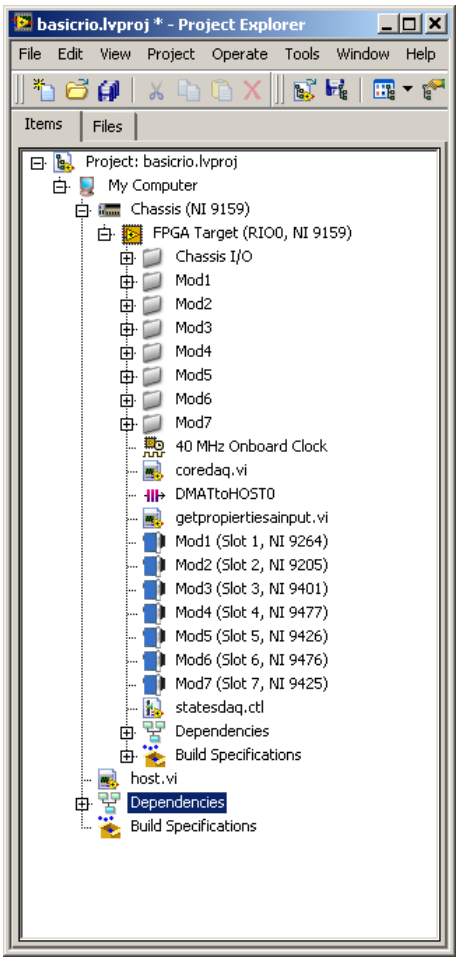

Fig. 8 cRIO LabVIEW Project

Fig. 8 show a project example with a compactRIO system NI 9159. This cRIO system is connected in the development computer using a PCIe with an MXIe interface. The NI9159 contains an FPGA (Virtex 5) and 7 input/output modules connected. Detailed information on how to use LabVIEW with FPGAs is available in [RD7] [RD8] [RD9] [RD10].

Once the LabVIEW project is created the DAQ application following the design rules can be built. These design rules apply to the VI to be developed for the FPGA only. The implementations of VIs for host computer have to follow the standard procedures when developing code for LabVIEW for Windows.

3.3 Design Rules for cRIO

The FPGA VI must contain a set of terminals that are mandatory regardless of the application. These terminals are presented in different colours and are used in order to identify clearly different functionalities. Some terminals need a default value because they will be read when the FPGA is not running yet. Independently of this the FPGA code description some additional rules described later in this document have to be met.

The cRIO design rules have been defined considering that the main objectives are:

a) The implementation of analog and digital I/O oriented applications for sampling rates below 250kS/s
b) The implementation of analog input waveform oriented applications

3.3.1 Platform Identification

Platform: Indicator register. Enum U8. This element is the register used to identify the hardware platform in use. The values for this terminal are shown in Table 2. The value of this terminal is read by the software driver when the bitfile has been downloaded to the FPGA but is not running.

Table 2 Values for Platform indicator

Platform	Value
FlexRIO	0
cRIO	1
R-Series	2

3.3.2 Mandatory Resources for a cRIO Design

Fig. 9 shows the basic resources to be added in a LabVIEW for FPGA design for cRIO. The meaning and functionality of the different terminals are explained below.

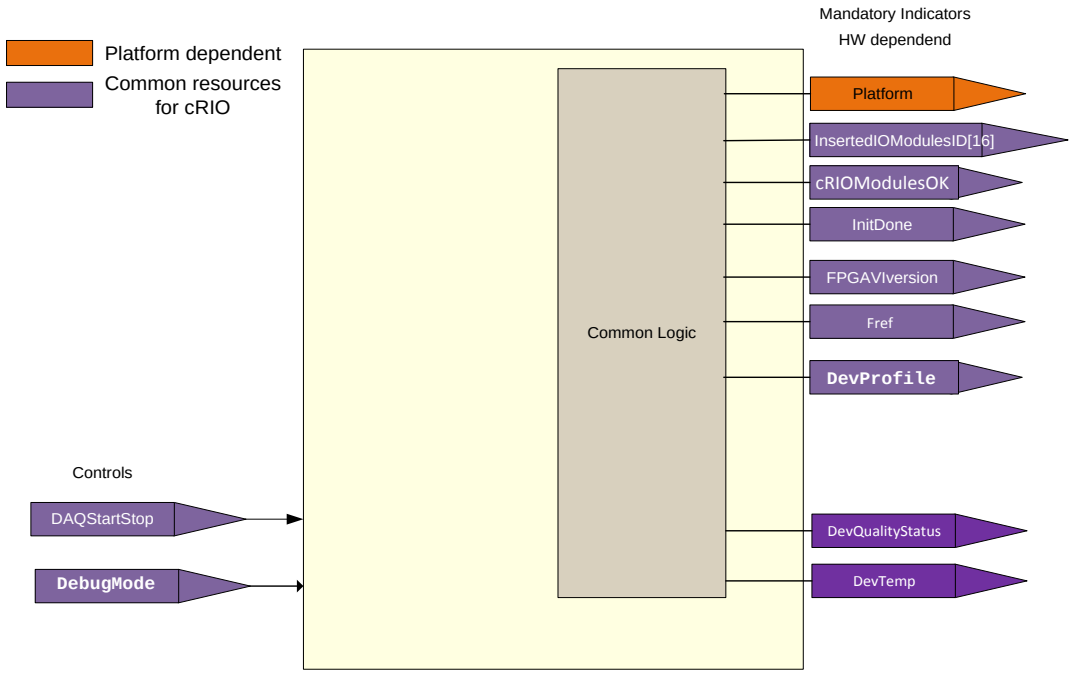

Fig. 9 Common terminals in the VI for cRIO

FPGAVIversion: U8 array indicator**.** This indicator will contain the version of the VI, which is checked by the software driver. The array only uses two elements.

InitDone: Boolean register indicator. This indicator is used to indicate that the FPGA and modules are correctly initialised. Zero means that the FGPA is not ready and 1 mean that the FPGA is ready. The designer should define when the FPGA and the I/O elements are ready to work checking the information provided by the manufacturer. The FPGA designer has to follow the specific steps defined for the I/O modules to execute the initialization.

Table 3 Values for Boolean Initdone indicator

Initdone	Value
Correct	True
Incorrect	False

InsertedIOModulesID[16]: Indicator Array of U16. Each position contains the Module ID as defined by National Instruments [RD11] and shown in Table 4. The size of this array should match the size of the chassis used.

Table 4: Module ID for cRIO

Module	ExpectedIOModuleID	I/O resources
NO Module	0x0000	No module installed
NI9205	0x712a	16/32 analog inputs
NI9264	0x 745C	16 analog outputs
NI9401	0x7130	TTL 8 I/O
NI9425	0x712F	24 V, Sinking Digital Input, 32 Ch Module
NI9426	0x736A	32-Channel, 24 V, 7 µs Sourcing Digital Input Module
NI9476	0x7133	24 V, Sourcing Digital Output, 32 Ch Module
NI9477	0x71CB	60 V, Sinking Digital Output, 32 Ch Module

cRIOModulesOK: Boolean Indicator. The expected modules and the modules installed matches.

Fref: U32 indicator. The indicator will contain the reference clock used for the sampling rate acquisition. This frequency is understood as the sampling rate of the data acquisition system.

DevQualityStatus: U8 indicator. This indicator informs the software driver about the possible errors in the signal conditioning or other possible situations.

DevTemp: I16 indicator. This indicator contents the temperature of the RIO's FPGA.

DevProfile: U8 indicator**.** This indicator is used to determine the kind of application implemented in the FPGA. The value specified here is very important because it defines the resources that mandatory will be searched and the optional resources used. The meaning of profile is different in the different platforms. If DevProfile is equals to 0 the implementation contains a design for analog waveform oriented data acquisition. Then, the resources defined for that profile are mandatory. For this profile waveform output generation, digital and analog point by point I/O are optional. If DevProfile is equals to 1 the profile Point By Point data acquisition is implemented. Table 5 and Table 6 summarize the mandatory and optional resources for the profiles. In the case of Point by Point acquisition at least one of the optional elements must be implemented.

Table 5: Values for DevProfile indicator

DevProfile	Info
0	Data acquisition
1	Point by Point acquisition (PBP)

Table 6: Resources for data acquisition profile (cRIO)

Resources	Info
Common	Mandatory
Data acquisition	Mandatory
Analog Input	Mandatory
Analog Output	Optional
Aux Analog Input	Optional
Aux Analog Output	Optional
Digital Output	Optional
Aux Digital Output	Optional
Digital Input	Optional
Aux Digital Input	Optional
DDS Waveform Generation	Optional

Table 7: Resources for point by point (PBP) acquisition profile (cRIO)

Resources	Info
Common	Mandatory
Data acquisition	forbidden
Analog Input	Optional
Analog Output	Optional
Aux Analog Input	Optional
Aux Analog Output	Optional

Resources	Info
Digital Output	Optional
Aux Digital Output	Optional
Digital Input	Optional
Aux Digital Input	Optional
DDS Waveform Generation	Optional

DAQStartStop: Control register Boolean type. This element is the register used to start and stop the data acquisition/generation in the RIO device. This terminal will start data acquisition/generation process in all the FPGA resources.

DebugMode: Control register Boolean type. This element is the register used to simulate the data acquired by the device. The behaviour of the simulation mode is defined by the developer.

3.3.3 Analog Signal Data Acquisition Profile (DMA-based)

3.3.3.1 *Mandatory resources for data acquisition profile.*

DMATtoHOSTNCh: U16 array indicator. This indicator has the information about the number of DMA channels implemented and channels allocated inside the different DMAs. The values of the different array elements are the number of channels. A group is defined as the set of channels included in one DMA.

DMATtoHOSTFrameType: U8 array indicator. This array has the same dimension size that DMATtoHOSTNCh. Every element in the array contains the data format used for the DMA data. The possible values for cRIO are: 0 to Format A and 1to Format B.

Table 8: Possible values for an element in DMATtoHOSTFrameType array

DMATtoHOSTFrameType [index]	Info
0	Format A
1	Format B

DMATtoHOSTSampleSize: U8 array indicator. This array has the same dimension size that **DMATtoHOSTNCh**. Every element in the array contains the number of bytes used per sample. In a specific design, all the channels included in DMA (DMA group) must have the same value of this parameter. The valid values are these shown in Table 9.

Table 9: Valid sample size in bytes

DMATtoHOSTSampleSize[n]	Info
0	Not valid
1	1 sample is one byte

DMATtoHOSTSampleSize[n]	Info
2	1 sample is 2 bytes
4	1 sample is 4 bytes
8	1 sample is 8 bytes

[Example for NI9159/NI9205]: DMATtoHOSTNCh [1] = {8} Eight channels for data acquisition. DMATtoHOSTFrameType [1]={0};

DMATtoHOSTSampleSize [1] = {4}. The number of samples per channels depends on the configuration of the cRIO module in the LabVIEW Projects. Raw configuration provides 2 bytes (I16) and calibrated provides 4 bytes because NI9205 represents the data in fixed point format with 21 bits, 5 for the integer part and 16 for the decimal one. These 21 bits are allocated in 4 bytes in the DMA as an I32 data.

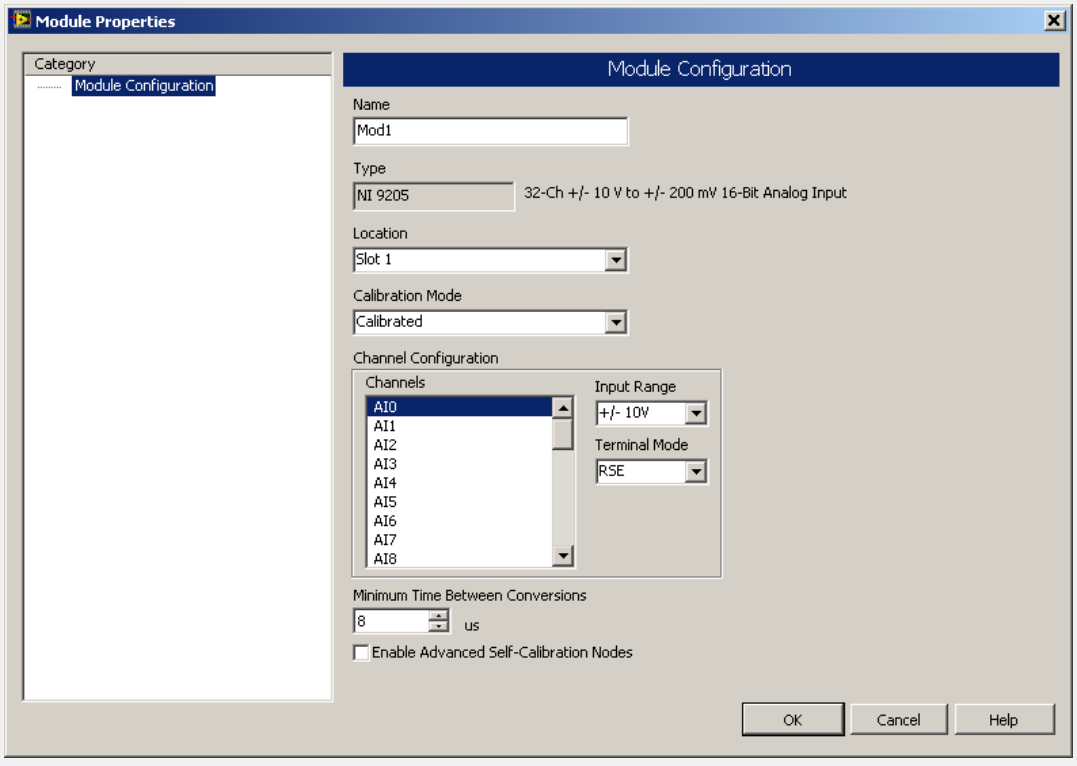

Fig. 10 NI-9205 Analog Input Module Configuration Window

DMATtoHOSTBlockNWords<n>: U16 array. This array has the same dimension and sizes that the previous ones. Each element contains the length of the block used in the data

acquisition. This terminal will inform to the software layer about the frame length. A frame is a set of samples of the different channels. The length of the block is defined as S.

DMATtoHOST<n>: This element is a target to host FIFO FPGA memory. This means that this memory is inside the FPGA implemented within the embedded RAM of the Virtes-5 LX110. This memory is used as a FIFO and it is always a 64-bit-wide FIFO connected to a DMA channel to send data to the HOST. The maximum number of FIFO DMAs is 3 for cRIO devices. If you have more than one DMA channel there will be as many DMATtoHOST elements as DMAs up to 3. The identification has to be correlative. Each DMA channel will send data acquired from a group of channels. Every DMA is a DMA group.

DMATtoHOSTSamplingRate<n>: Control register. U16. There must be as many "**DMATtoHOSTSamplingRate**" controls as DMAs used to pass acquired data to the CPU. The data acquired will be packaged into groups of channels and then sent through each DMA. The label used must be enumerated from 0 to I_{max}-1. I_{max} is the maximum number of DMA channels available for the cRIO device (3). If the design includes more than one DMA, there will be a set of controls that we can define as DMATtoHOSTSamplingRate0 [0..I_{max}-1]. These terminals control the sampling frequency from DMA group 0 to I-1.

DMATtoHOSTEnable<n>: Boolean register control. There are as many **DMATtoHOSTEnable** controls (**DMATtoHOSTEnable** [0.. I_{max}-1]) as DMA groups. The data of the group are acquired if this control is set to true.

DMATtoHOSTOverflows: U16 indicator. Each bit of this indicator will show the status of each of the device's possible DMAs. The status will be either Correct (0) or Overflow (1).

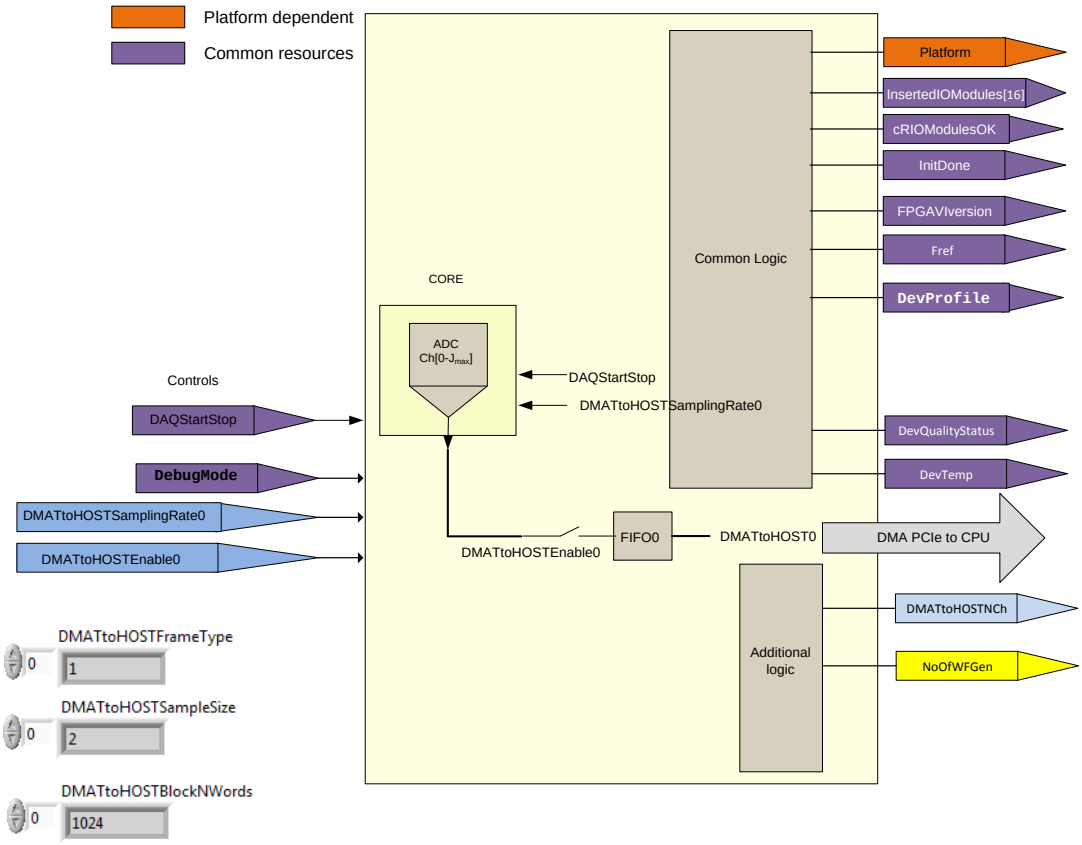

Fig. 11 coreDAQ. Minimum element for implementing data acquisition in a cRIO device

3.3.3.2 *Data format in the DMA for Data acquisition profile.*

The data acquisition profile is oriented for the acquisition of analog input channels and supports different formats in the data stream sent to the HOST using the DMA.

3.3.3.2.1 NI9159/NI9205

Every NI9205 provides 16/32 analog inputs. The NI9205 can be configured in raw format or in the calibrated one. One channel sample uses an I16 in raw mode and 21 bits in calibrated one. This point should be considered when defining the bytes used for sample.

3.3.3.2.2 Format A: DAQ samples

The data in the DMA must be formatted according to the following rules:

- The number of channels N is variable between 1 and 256. N is configured in the FPGA for every DMA using the corresponding DMATtoHOSTNCh [i] element.

- W: Bytes used per sample. W=2 for instance for NI9205 in raw format or W=4 in calibrated one. All channels in the DMA use the same W. The valid values for bytes used per samples are 1, 2, 4 or 8. W is specified in the DMATtoHOSTSampleSize array.
- S is the number of samples S in a block. Every block has a length of U64 data with S samples (the number of channels included is defined with N).S must be an integer number multiple of N*W/4 . This value is specified using the DMATtoHOSTBlockNWords array.
- The acquired data must be always encapsulated in 64-bit words of the DMA FIFO.

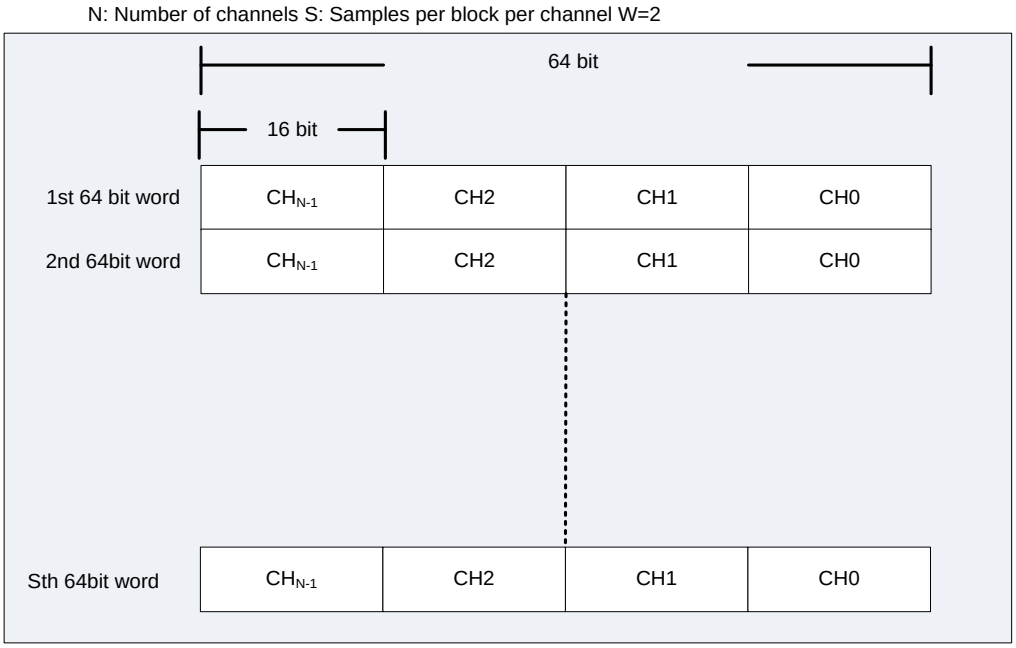

Fig. 12 Data organization in the DMA. Example for N=4

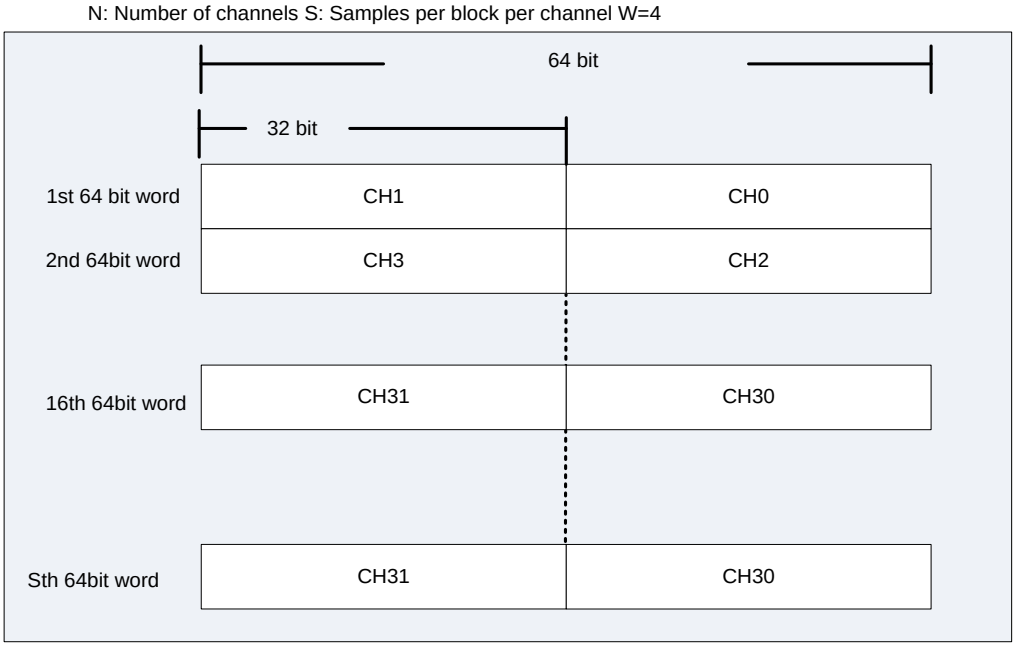

Fig. 13 Data organization in the DMA. Example for N=32

3.3.3.3 *Optional resources*

The optional resources checked in this profile are listed below.

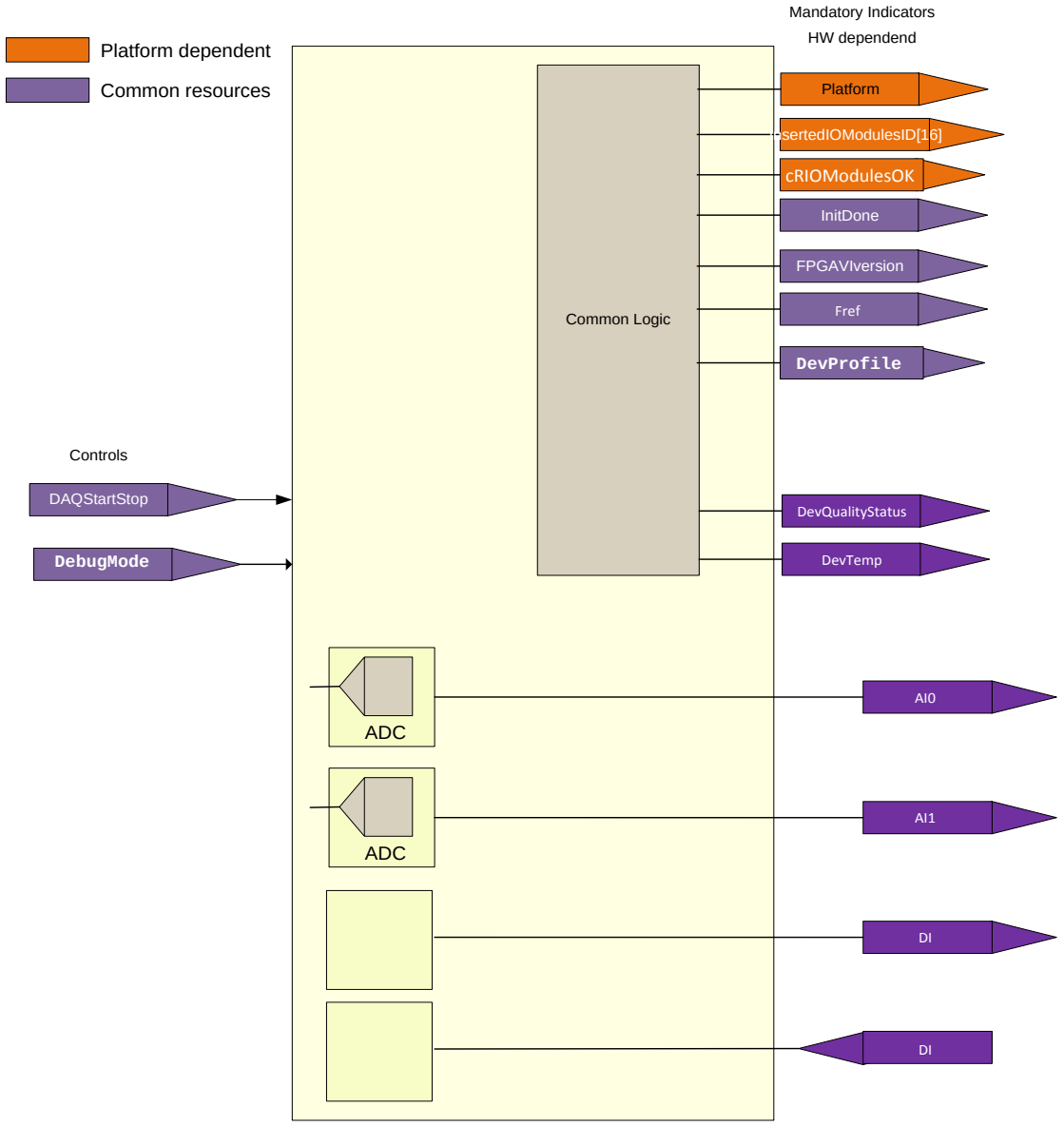

Fig. 14 Mandatory Resources and Some Non-Mandatory Resources

3.3.3.3.1 Analog inputs

AI<x>: I32 indicator. The FPGA LabVIEW programmer can add read-only registers (indicators) with the last sample acquired from an analog input. This indicator will be updated at the sampling rate programmed for the channel. The nomenclature for naming the indicator will be "AI" followed by the number of the channel. The maximum number of AI terminals is 64. The AI<x> are only used if there are NI9205 modules installed.

3.3.3.3.2 Auxiliary analog inputs

auxAI<x>: I32 indicator. The FPGA designer can include indicators identified as auxAI<N> with LabVIEW I32 data type representing any internal variable in the FPGA. For instance, you can acquire one sample from adapter module analog input channels (I16), operate the data and connect the result to an I32 terminal labelled as auxAI0. The maximum number is 16.

3.3.3.3.3 Analog Output

AO<x>: I32 indicator. These terminals will be connected to the physical I/O nodes available in I/O module. The maximum number of analog outputs is 32.

3.3.3.3.4 Auxiliary analog output

auxAO<x>: I32 control. The FPGA designer can include controls identified as auxAO<x> with LabVIEW I32 data type representing any internal variable in the FPGA. The maximum number de auxAO is 16.

3.3.3.3.5 Digital input/output

DO<n>: Boolean control. The FPGA designer can include controls identified as DO<n>. These controls will be connected to physical digital outputs in an I/O module. The maximum number of DO is 96.

DI<n>: Boolean indicator. The FPGA designer can include indicators identified as DI<n>. The maximum number is 96.

auxDO<n>: Boolean control. The FPGA designer can include controls identified as DO<n>. These controls will be connected to internal FPGA signals. The maximum number is 16.

auxDI<n>: Boolean indicator. The FPGA designer can include indicators identified as auxDI<n>. These indicators will be connected to internal FPGA signals in a FlexRIO adapter module. The maximum number is 16.

3.3.3.3.6 Signal generator

SGNo: U8 indicator. This indicator is initialised with the number of waveform generators included in the design. A null value means no signal generator implemented. The values allowed are from 0 to 16.

In the cRIO device, the user can add an element to implement signal generation using the analog outputs. The templates provide a signal generator implemented with direct digital synthesis (DDS) technique. In this method, the FPGA contains a memory with a predefined pattern. The terminals available to use this block are described in Table 10.

Table 10: Signal generator terminals

LabVIEW Terminal Name	Type	Functionality	Notes
SGFreq<n>	U32, Control	Frequency of the signal to be generated	The desired frequency (freq) in Hertz and the terminal follow this equation

LabVIEW Terminal Name	Type	Functionality	Notes
			$SGFreq = freq \times \frac{2^{32}}{Frefo}\ x$ SGUpdateRate Frefo is the frequency used in the output generator
SGAmp<n>	U16,Control	Amplitude of the signal to be generated	The value to be written in the terminal must be a value from 0 to 32767.
SGPhase<n>	U32, Control	Phase control for the signal	The terminal contains the phase shift
SGSignalType<n>	U8, Control	Signal type among DC, Sine, Triangular and Square	Enumerated value to select the signal needed
SGUpdateRate<n>	U32, control	Update rate frequency used for signal generation	The analog output is updated using a frequency equals to the SGUpdateRatexfrefo
SGFref<n>	U32, Indicator	Defines the reference frequency used by the signal generator	Defines the reference frequency used by the signal generator

3.3.3.4 *Summary of resources for cRIO DAQ profile*

Table 11 summarizes the terminals (control and indicators) used by data acquisition profile in cRIO platform. The DMA-based DAQ template has been implemented using these terminals.

Table 11: Summary of resources for cRIO DAQ profile

Terminal Name	Data type	Type	Detail	Information	Values	Initialized before run?
Platform	U8	Indicator	This terminal defines the form factor used in the FPGA implementation	Mandatory	0- FlexRIO 1- cRIO 2- R Series	YES

Terminal Name	Data type	Type	Detail	Information	Values	Initialized before run?
Mandatory resources for cRIO						
FPGAVIversion	Array U8	Indicator	Contains the VI version, 2 elements. One for MM major version, and the next one mm minor version. MM.mm	Mandatory	For instance 1.1 FPGAVIVersion[0]=1 FPGAVIVersion[1]=1	YES
InitDone	Boolean	Indicator	This terminal must be set to true when the FPGA is initialized	Mandatory	True=OK False=NOK	N/A
InsertedOModulesID	Array U16	Indicator	Numeric array of values indicating the c-Modules IDs detected	Mandatory	Defined by NI	NO
cRIOModulesOK	Boolean	Indicator	I/O Modules correctly detected	Mandatory		NO
Fref	U32	Indicator	Contains the Reference clock of the FPGA for sampling rate	Mandatory		YES
DevQualityStatus	U8	Indicator	This indicator will show the status of the acquisition	Mandatory		NO
DevTemp	I16	Indicator	This indicator	Mandatory		NO

Terminal Name	Data type	Type	Detail	Information	Values	Initialized before run?
			will show the temperature of the FPGA			
DevProfile	U8	Indicator	This indicator defines the implementation in the FPGA (DAQ, Image, etc.)	Mandatory		YES
DebugMode	Boolean	Control	If debug is true the FPGA will simulate the acquired data. Otherwise, physical signals are acquired	Mandatory		NO
DAQStartStop	Boolean	Control	This terminal must be set to true to start data acquisition	Mandatory		
Specific Terminals for cRIO DAQ profile						
DMATtoHOSTNCh	Array U16	Indicator	Describes the number of DMAs implemented in the FPGA. The array must be initializes with the number of channels available in each DMA.	Mandatory	n ={0 .. 2}	YES

Terminal Name	Data type	Type	Detail	Information	Values	Initialized before run?
DMATtoHOSTFrameType	Array U8	Indicator	Describes the frame type used in the DMA frame	Mandatory	n ={0 .. 2}	YES
DMATtoHOSTSampleSize	Array U8	Indicator	Size in bytes for the channel sample	Mandatory	n ={0 .. 2}	YES
DMATtoHOSTBlockNWords	Array U16	Indicator	Length of the block	Mandatory	n ={0 .. 2}	YES
DMATtoHOST<n>	FIFO	DMA target to HOST	FIFO memory in the FPGA	Mandatory	n ={0 .. 2}	N/A
DMATtoHOSTSamplingRate<n>	U16	Control	Integer number obtained as Sampling rate/Fref	Mandatory	n ={0 .. 2}	N/A
DMATtoHOSTEnable<n>	Boolean	Control	Enable or disable write to DMA FIFO	Mandatory	n ={0 .. 2}	N/A
DMATtoHOSTOverflows	U16	Indicator	Status of the different DMA FIFO	Mandatory		
Optional Resources for this profile						
AI<n>	I32	Indicator	Digital sample	Optional	n ={0 .. 255}	
auxAI<n>	I32	Indicator	Auxiliary internal FPGA variables	Optional	n ={0 .. 255}	
AO<n>	I32	Indicator	Digital Sample	Optional	n ={0 .. 255}	
auxAO<n>	I32	Control	Auxiliary internal FPGA variables	Optional	n ={0 .. 255}	N/A
AOEnable<n>	Boolean	Control	Enable or Disable analog output		n ={0 .. 255}	

Terminal Name	Data type	Type	Detail	Information	Values	Initialized before run?
DO<n>	Boolean	Control	Digital line	Optional	n ={0 .. 255}	
auxDO<n>	Boolean	Control	Digital line	Optional	n ={0 .. 255}	
DI<n>	Boolean	Indicator	Digital line	Optional	n ={0 .. 255}	
auxDI<n>	Boolean	Indicator	Digital line	Optional	n ={0 .. 255}	
SGNo	U8	Control	Number of waveform generators	Optional	n ={0 .. 255}	YES
SGSignalType<n>	U8	Control	Signal shape to be generated	Optional	User selectable	YES
SGFreq<n>	U32	Control	DSS accumulator increment	Optional	n ={0 .. 255}	YES
SGAmp<n>	U16	Control		Optional	n ={0 .. 255}	YES
SGPhas<n>	U32	Control	Phase control	Optional	n ={0 .. 255}	YES
SGUpdateRate<n>	U32	Control	Update rate	Optional	n ={0 .. 255}	N/A
SGFref<n>	U32	Indicator	Reference frequency	Optional	n={1..255}	

3.3.4 Point by Point Acquisition Profile

This profile is oriented to the implementation of analog and digital I/O operations.

3.3.4.1 *Mandatory resources for point by point I/O profile*

SamplingRate<n>: U16 control. This terminal allows programming the sampling rate. The sampling rate value in S/s is the Fref value divided by SamplingRate<n>. This sampling rate is used for controlling the data acquisition/generation of the different I/O elements in the cRIO chassis.

3.3.4.2 *Optional resources*

The following resources are optional and can appear or not in the design.

3.3.4.2.1 Analog inputs

AI<x>: I32 indicator. The FPGA LabVIEW programmer can add read-only registers (indicators) with the last sample acquired from an analog input. This indicator will be updated at the sampling rate programmed for the channel. The nomenclature for naming the indicator

will be "AI" followed by the number of the channel. The maximum number of AI terminals is 32*14=384 (14 slots with NI9205). The AI<x> are only used if there are NI9205 modules installed.

3.3.4.2.2 Auxiliary analog inputs

auxAI<x>: I32 indicator. The FPGA designer can include indicators identified as auxAI<N> with LabVIEW I32 data type representing any internal variable in the FPGA. For instance, you can acquire one sample from adapter module analog input channels (I16), operate the data and connect the result to an I32 terminal labelled as auxAI0. Maximum number of auxAI is 16.

3.3.4.2.3 Analog Output

AO<x>: I32 control. These terminals will be connected to the physical I/O nodes available in I/O module. The maximum number of analog outputs is 16 times the number of modules, 256.

3.3.4.2.4 Auxiliary analog output

auxAO<x>: I32 control. The FPGA designer can include controls identified as auxAO<x> with LabVIEW I32 data type representing any internal variable in the FPGA. The maximum number de auxAO is 256.

3.3.4.2.5 Digital input/output

DO<n>: Boolean control. The FPGA designer can include controls identified as DO<n>. These controls will be connected to physical digital outputs in an I/O module. The maximum number of DO is 256.

DI<n>: Boolean indicator. The FPGA designer can include indicators identified as DI<n>. The maximum number is 256.

auxDO<n>: Boolean control. The FPGA designer can include controls identified as DO<n>. These controls will be connected to internal FPGA signals. The maximum number is 256.

auxDI<n>: Boolean indicator. The FPGA designer can include indicators identified as auxDI<n>. These indicators will be connected to internal FPGA signals in a cRIO adapter module. The maximum number is 256.

3.3.4.2.6 Signal Generator

See signal generator description in data acquisition profile.

3.3.4.3 *Summary of resources for cRIO Point by Point profile*

Table 12 summarizes the terminals (control and indicators) used by data acquisition profile in cRIO platform. The Point by Pont DAQ template has been implemented using these terminals.

Table 12: Summary of resources for PBP Profile

Terminal Name	Data type	Type	Detail	Information	Values	Initialized before run?
Platform	U8	Indicator	This terminal defines the form factor used in the FPGA implementation	Mandatory	0- FlexRIO 1- cRIO 2- R Series	YES

Terminal Name	Data type	Type	Detail	Information	Values	Initialized before run?
Common Terminals for cRIO						
FPGAVIversion	Array U8	Indicator	Contains the VI version, 2 elements. One for MM major version, and the next one mm minor version. MM.mm	Mandatory	For instance 1.1 FPGAVIVersion[0]=1 FPGAVIVersion[1]=1	YES
InitDone	Boolean	Indicator	This terminal must be set to true when the FPGA is initialized	Mandatory	True=OK False=NOK	N/A
InsertedOModulesID	Array U16	Indicator	Numeric array of values indicating the c-Modules IDs detected	Mandatory	Defined by NI	NO
cRIOModulesOK	Boolean	Indicator	I/O Modules correctly detected	Mandatory		NO
Fref	U32	Indicator	Contains the Reference clock of the FPGA for sampling rate	Mandatory		YES
DevQualityStatus	U8	Indicator	This indicator will show the status of the acquisition	Mandatory		NO
DevTemp	I16	Indicator	This indicator will show the temperature of the FPGA	Mandatory		NO
Devprofile	U8	Indicator	This indicator defines the implementation in the FPGA (DAQ, Image, etc.)	Mandatory		YES
DebugMode	Boolean	Control	If debug is true the FPGA will simulate the acquired data. Otherwise, physical signals are acquired	Mandatory		NO
DAQStartStop	Boolean	Control	This terminal must be set to true to start data acquisition	Mandatory		
Specific Terminals for Point by Point acquisition profile						
SamplingRate<n>	U16	Control	Integer number obtained as Sampling rate/Fref	Mandatory	n ={0 .. 2}	
Optional Resources for Point by Point profile						
AI<n>	I32	Indicator	Digital sample	Optional	n ={0 .. 255}	
auxAI<n>	I32	Indicator	Auxiliary internal FPGA variables	Optional	n ={0 .. 255}	
AO<n>	I32	Indicator	Digital Sample	Optional	n ={0 .. 255}	

Terminal Name	Data type	Type	Detail	Information	Values	Initialized before run?
auxAO<n>	I32	Control	Auxiliary internal FPGA variables	Optional	n ={0 .. 255}	
AOEnable<n>	Boolean	Control	Enable or Disable analog output		n ={0 .. 255}	
DO<n>	Boolean	Control	Digital line	Optional	n ={0 .. 255}	
auxDO<n>	Boolean	Control	Digital line	Optional	n ={0 .. 255}	
DI<n>	Boolean	Indicator	Digital line	Optional	n ={0 .. 255}	
auxDI<n>	Boolean	Indicator	Digital line	Optional	n ={0 .. 255}	
SGNo	U8	Control	Number of waveform generators	Optional	n ={0 .. 255}	YES
SGSignalType<n>	U8	Control	Signal shape to be generated	Optional	User selectable	YES
SGUpdateRate<n>	U32	Update rate frequency used for signal generation	The analog output is updated using a frequency equals to the SGUpdateRatexfrefo	SGUpdateRate<n>	U32, control	Update rate frequency used for signal generation
SGFreq<n>	U32	Control	DSS accumulator increment	Optional	n ={0 .. 255}	YES
SGAmp<n>	U16	Control		Optional	n ={0 .. 255}	YES
SGPhase<n>	U32	Control	Phase control	Optional	n ={0 .. 255}	YES
SGFref<n>	U32	Indicator	Reference frequency	Optional	n={1..255}	

3.4 cRIO Example Templates

Two different templates had been created to cover the two general uses intended for the cRIO platform, one of them is intended to be used as an example of basic input/output and the other implementation is intended to be used for analog input waveform oriented applications. Each of them implements one of the aforementioned profiles respectively and the two follow the design rules for cRIO devices and implement some additional I/O to illustrate the use cases. This section additionally describes the characteristics of the I/O adapter modules and proposes an interconnection between pairs of modules to test the functionalities of the applications without any external system connected to the NI-9159 chassis.

3.4.1 cRIO Basic Requirements for the Examples Provided

This section describes the system requirements for running the examples using a NI CompactRIO chassis and seven NI CompactRIO I/O Modules hosted in the chassis. NI 9159, 14-slot CompactRIO Chassis, LX 110 FPGA, MXIe

- NI 9205 32-Ch ±200 mV to ±10 V, 16-Bit, 250 kS/s AI Module

- NI 9264 16-Ch ±10 V, 16-Bit, 25 kS/s Analog Output Module
- NI 9477 32-Ch 24 V, 8 µs, Sinking DO Module
- NI 9425 32-Ch 24 V, 7 µs, Sinking DI Module
- NI 9476 32-Ch 24 V, 500 µs, Sourcing DO Module
- NI 9426 32-Ch 24 V, 7 µs, Sourcing DI Module
- NI 9401 8-Ch, 5 V/TTL High-Speed Bidirectional Digital I/O Module

The module placement for running the examples is depicted in Fig. 15.

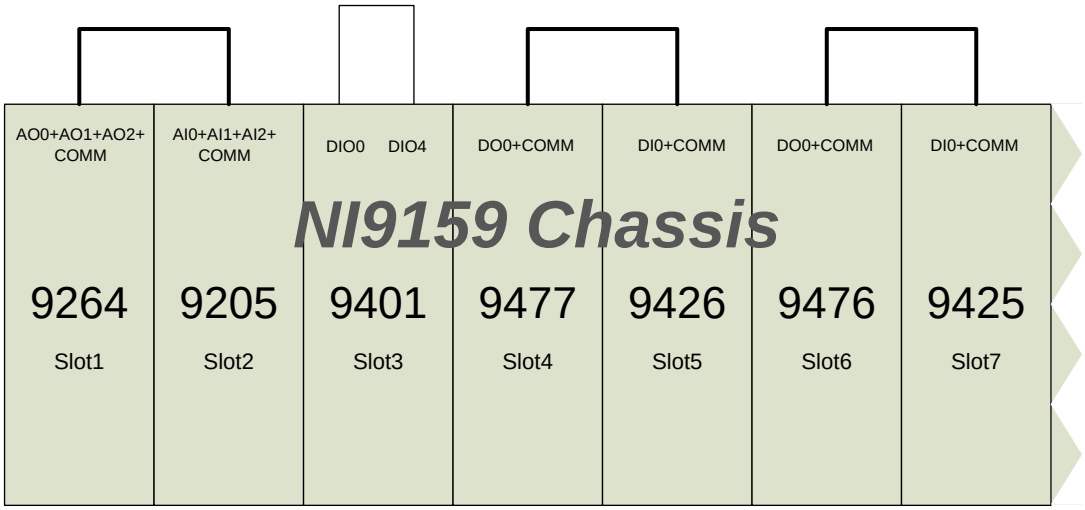

Fig. 15 NI9159 Chassis Generic Architecture

3.4.2 Module Identification in the Chassis

A NI9159 chassis [RD12] has 14 slots for connecting compact RIO modules. The number scheme of the modules inserted in each slot are identified from Module 1 to Module 14 numbered from left to right. The examples described in these sections are implemented with 7 modules.

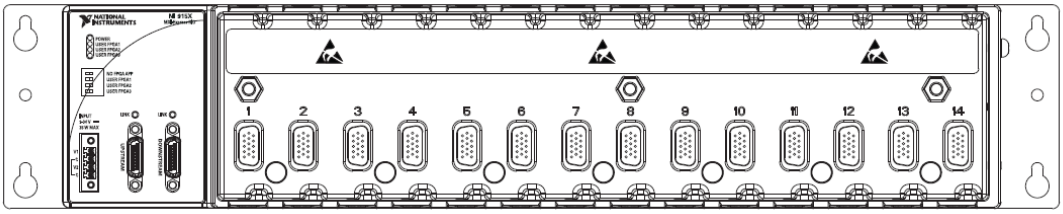

Fig. 16 Chassis NI9159 and Slot Numbering Schema

3.4.3 Module Description and Signal Interconnections

The subsequent section describes the modules used in the system and the signal interconnection among them supporting the connections schema depicted in Fig. 15.

3.4.3.1 *NI9205 Analog Input Module*

This module is used to measure analog input (AI) signals. The first three AI ports of NI9205 module is connected to the first three ports of the NI9264 module (Module 1)

- AO_0 NI9264 $Module_1$ to AI_0 $Module_2$
- AO_1 NI9264 $Module_1$ to AI_1 $Module_2$
- AO_2 NI9264 $Module_1$ to AI_2 $Module_2$

The COMM port of $Module_1$ and $Module_2$ is interconnected. Aforementioned connections are Referenced Single-Ended (RSE mode) as depicted in Fig. 18. The figures shown in Table 13 are the most relevant characteristics extracted from the NI9205 Datasheet [RD13]. Fig. 17 depicts the signal connector pins of the module.

Table 13 NI9205 Module Relevant Characteristics

NI9205 Module relevant characteristics (extracted from datasheet)	
Conversion Time	R Series Expansion chassis: 4.50µs
	All other chassis: 4.00µs

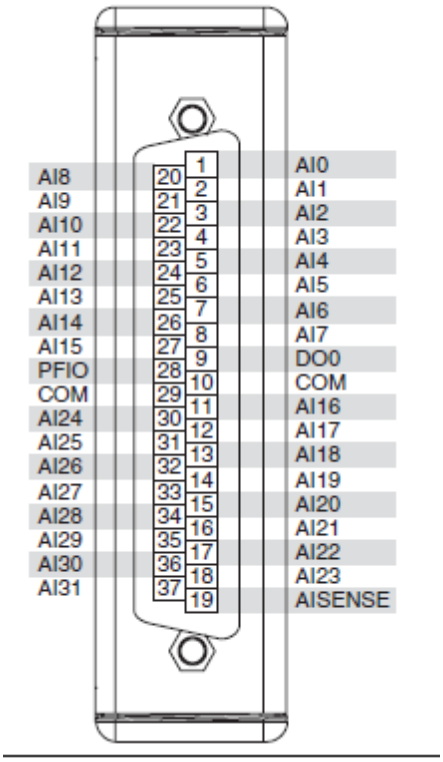

Fig. 17 NI9205 Signal Connector

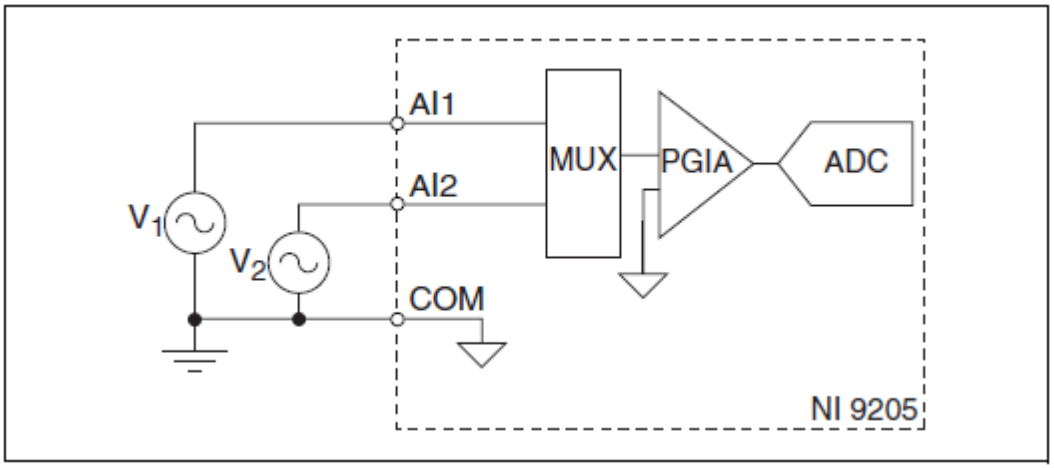

Fig. 18 Signal connections in RSE mode

3.4.3.2 *NI9264 Analog Output Connection*

The NI9264 is used to generate analog output signals. The connection to Module$_2$ is described in Table 15. The connector and the signals connection is detailed via Fig. 19 and Fig. 20, in

addition the main figures of this module extracted from the datasheet [RD14] are presented in Table 14.

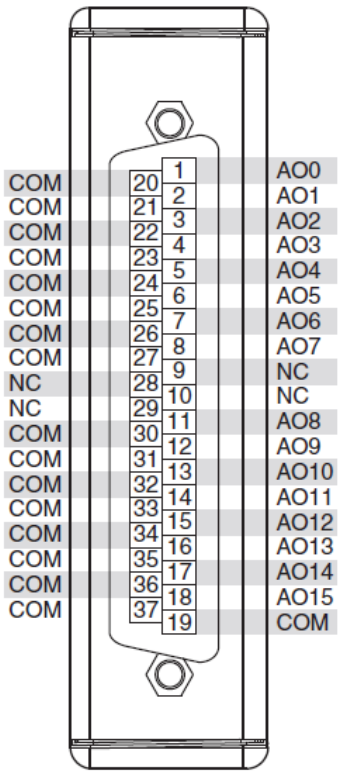

Fig. 19 Signals in the NI9264 analog output module

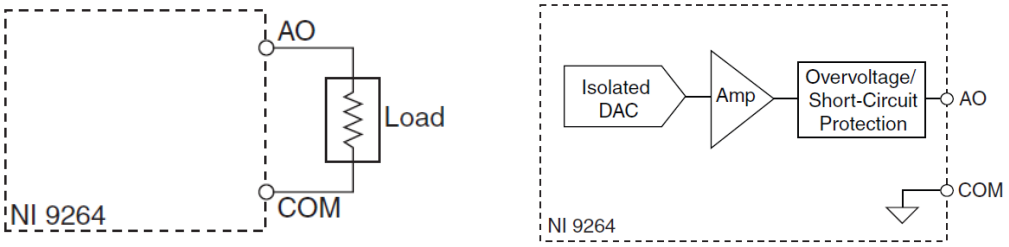

Fig. 20 Connection for analog output channels.

Table 14 NI9264 relevant characteristics

NI9264 Module relevant characteristics (extracted from datasheet)		
Settling time (100pF load, to 1 LSB)	20V Step	20µs
	1V Step	15 µs

NI9264 Module relevant characteristics (extracted from datasheet)		
	0.1V Step	13 µs
Update Time	1 Channel	3.1 µs min.
	2 Channels	5.3 µs min.
	3 Channels	7.5 µs min.
	16 Channels	37 µs min.

Table 15 Interconnection between NI9264 and NI9205

NI9264 Analog output channel	NI9205 Module/channel
$Module_1$ /$Channel_0$	$Module_2$ /$Channel_0$
$Module_1$ /$Channel_1$	$Module_2$ /$Channel_1$
$Module_1$ /$Channel_2$	$Module_2$ /$Channel_2$

3.4.3.3 *NI9401 Digital Input/Output*

The NI9401 is an 8-Channel Digital Input/Output TTL Module. The $Port_0$ and the $Port_4$ is externally interconnected. This module requires configuration to set the line direction of the two pairs of 4 ports. The connector description, the signal connection and the basic characteristic extracted from its datasheet [RD15] are shown in Fig. 21, Fig. 22 and Table 15.

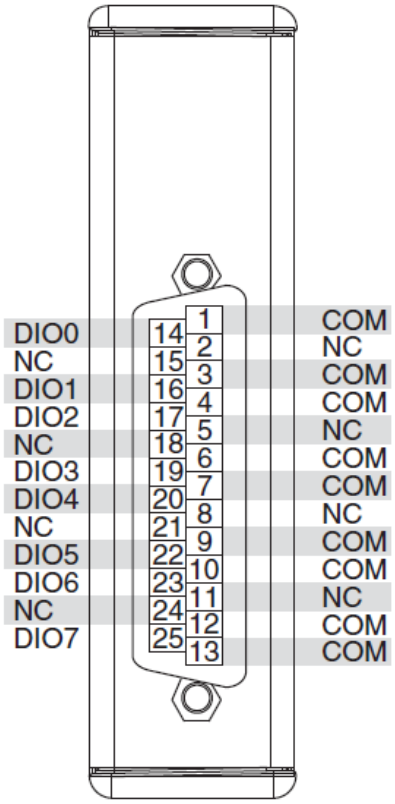

Fig. 21 Signal connector for NI9401

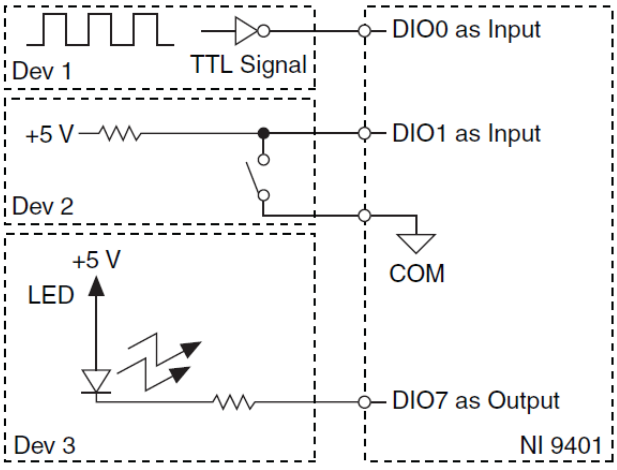

Fig. 22 Signal connections for Digital Input and Output in NI9401

Table 16 NI9401 relevant characteristics

NI9401 Digital Input/Output Module relevant characteristics (extracted from datasheet)

NI9401 Digital Input/Output Module relevant characteristics (extracted from datasheet)		
Maximum input signal switching frequency by number of input channels, per channel	8 input channels	9 MHz
	4 input channels	16 MHz
	2 input channels	30 MHz
Maximum output signal switching frequency by number of output channels with an output load of 1mA, 50pF, per channel	8 output channels	5 MHz
	4 output channels	10 MHz
	2 output channels	20 MHz

3.4.3.4 *NI9477 Sinking Digital Output module*

The NI9477 Sinking Digital Output module [RD16] can be used to interface sourcing digital actuators. It is connected to the $Module_5$ NI9426.

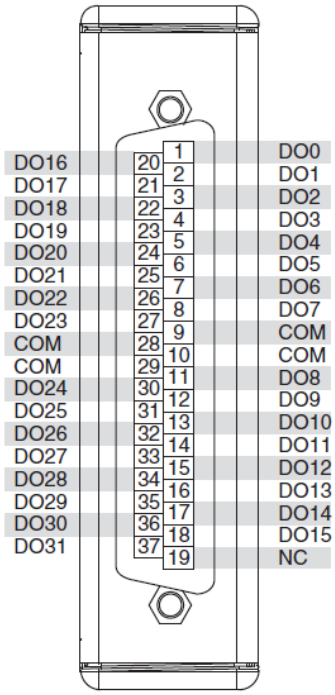

Fig. 23 NI9477 32 channel digital output Signal Connector

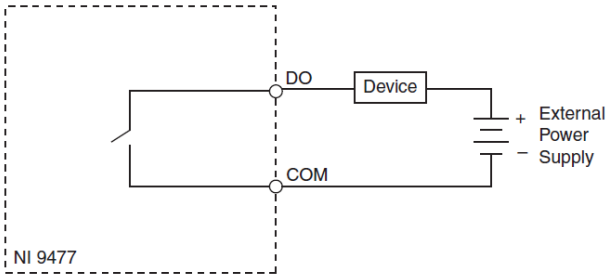

Fig. 24 Connection of an external device to NI9477

Table 17 NI9477 Relevant Characteristics

NI9477 0-60V Sinking Digital Output Module relevant characteristics (extracted from datasheet)	
Maximum Update Rate	8 µs max.
Propagation Delay	1 µs max.

3.4.3.5 *NI9426 Sourcing Digital Input Module*

The NI9426 is a sourcing digital input module [RD17]. One digital output (DO) channel of the NI9477 $Module_4$ is connected to a digital input (DI) of the NI9426 $Module_5$. The connector is depicted in Fig. 25, the connection layout to a device is depicted in Fig. 26, and the most relevant characteristics are exposed in Table 18.

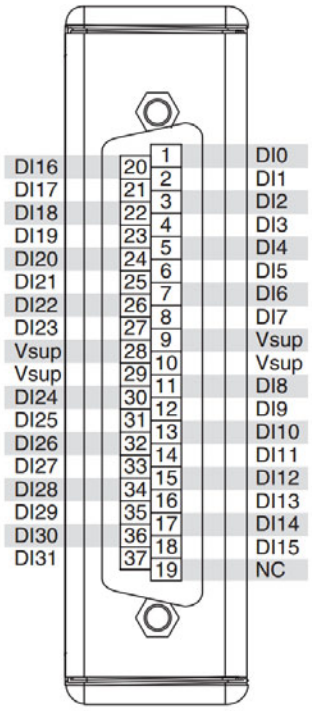

Fig. 25 Signals in the NI9426 digital input module

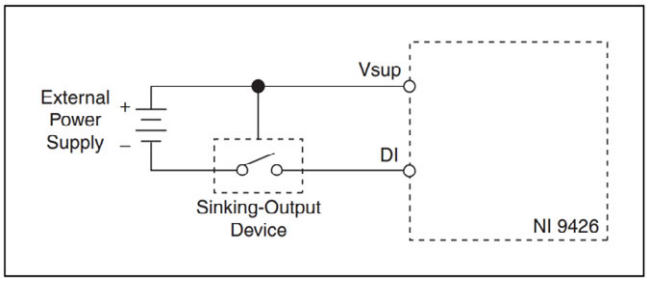

Fig. 26 Connecting a device to the NI9426

Table 18 NI9426 Relevant Characteristics

NI9426 32 channel 24V Sourcing Digital Input Module relevant characteristics (extracted from datasheet)	
Digital Logic Levels Input Voltages	OFF State ≥ (V_{sup} - 5 V)
	ON State ≤ (V_{sup} - 10 V)
Update/Transfer time	7 µs max.
Setup time	1 µs min.

3.4.3.6 *NI9425 Sinking Digital Input Module*

NI9425 is used as Digital Input (DI) module [RD18] and it is connected to the NI9476 Module$_6$. The connector is depicted in Fig. 25, the connection layout to a device is depicted in Fig. 28, and the most relevant characteristics are exposed in Table 19.

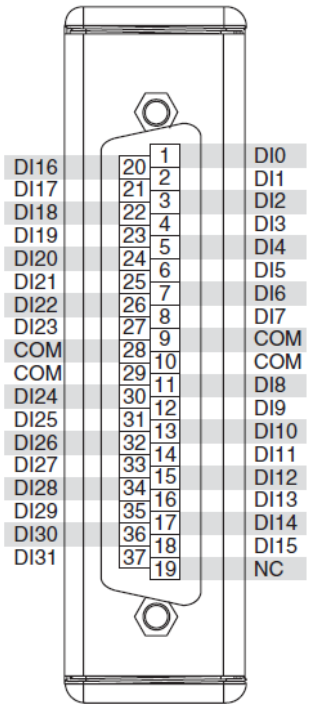

Fig. 27 Connection in NI9425 module

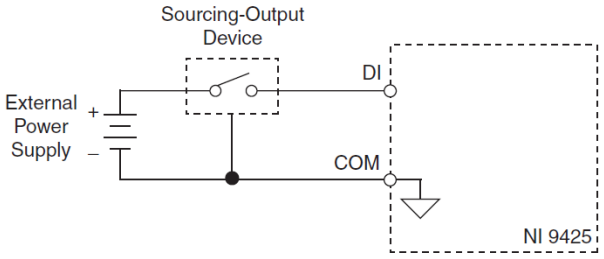

Fig. 28 Connecting a device to the NI9425

Table 19 NI9425 Relevant Characteristics

NI9425 32 channel 24V Sinking Digital Input Module relevant characteristics (extracted from datasheet)	
Digital Logic Levels Input Voltages	OFF State ≤ 5 V
	ON State ≥ 10 V
Update/Transfer time	7 µs max.
Setup time	1 µs min.

3.4.3.7 *NI9476 Sourcing Digital Output Module*

The NI9476 32-Channel 24V sourcing digital output module [RD19] can be used to interface to a 24V actuator. The connector is depicted in Fig. 29, the connection layout to a device is depicted in Fig. 30, and the most relevant characteristics are exposed in Table 20.

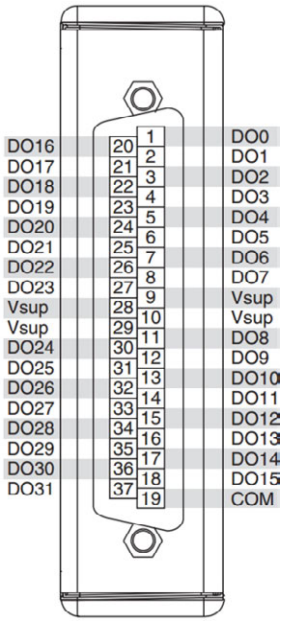

Fig. 29 NI9476 Signal Connector

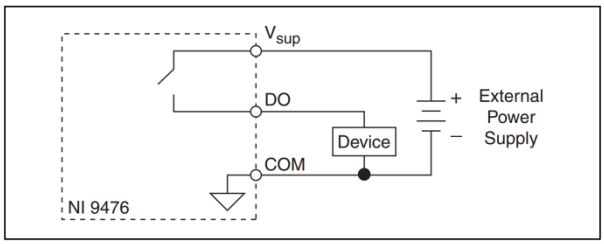

Fig. 30 Connection of a device to the NI9476

Table 20 NI9476 Relevant Characteristics

NI9476 32 channel 24V Sourcing Digital Output Module relevant characteristics (extracted from datasheet)	
Maximum Update Rate	40 µs max.
Propagation Delay	500 µs max.

3.4.4 System General Description

3.4.4.1 *General Block Diagram*

The two examples for cRIO architecture presented bellow have the same high level architecture depicted in Fig. 31.

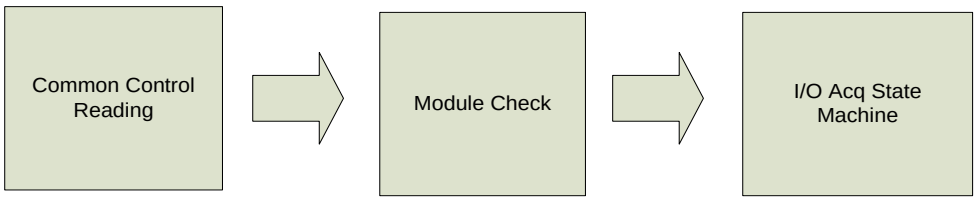

Fig. 31 Functional Architecture

3.4.4.2 *State Machine*

The system behaviours as a state machine composed by three main states, as represented in Fig. 32. *Initialization state* performs the system power up, checks the system configuration (all modules are present in their corresponding chassis slot and working properly). *Wait state* waits for operator click start button in the HMI. The *Operation State Machine* (*a.k.a. I/O Acquisition Loop*) performs the acquisition and generation of signals in a continuous loop. The system continues operation while the *DAQStartStop* remains true, if false the state machine returns to the *Initialization state*.

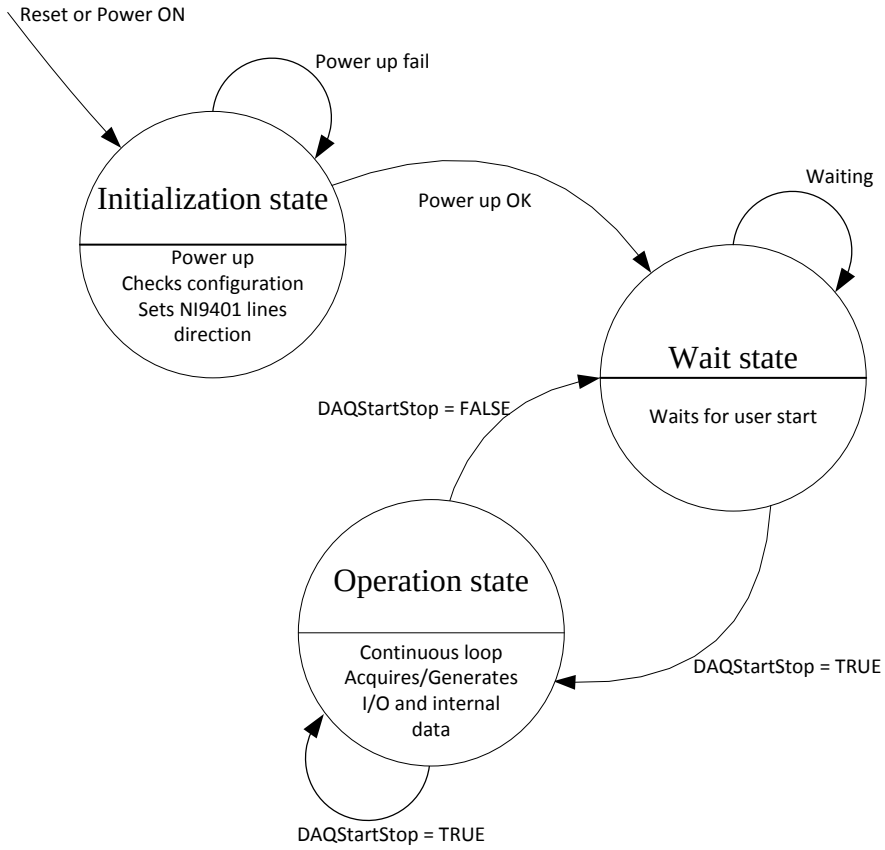

Fig. 32 State Machine

3.4.4.3 *Operation State: I/O Acquisition Loop*

This piece of logic implemented in the FPGA performs the main functionality of the modelled system. The concrete behaviour of each system will be described in the following sections.

3.4.4.4 *System Management: Host HMI*

The CompactRIO systems are based on reconfigurable FPGA chassis and I/O modules. The LabVIEW FPGA Module enables to translate this code directly to hardware. This process requires the compilation of the code to be synthesized to a bitfile (see Fig. 33).

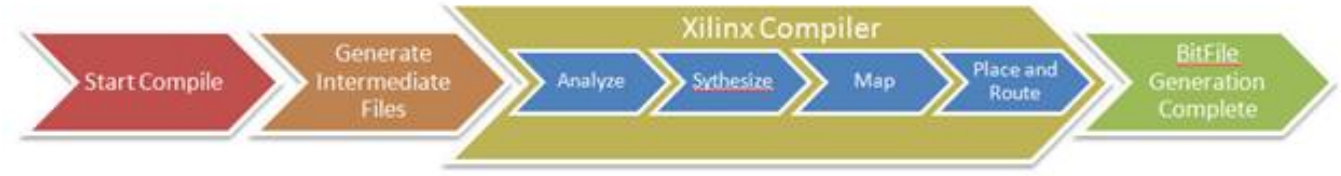

Fig. 33 LabVIEW FPGA Compilation Process

To ease the process of downloading the bitfile to the FPGA and run and monitor the application, a host HMI application has been developed using LabVIEW on Windows OS. Running this application, the bitfile is downloaded to the FPGA target of the chassis and the state machine (described in 3.4.4.2) is started. From this point, the application remains communicating with the FPGA code to monitor the status of the system. Fig. 34 depicts the front panel of a generic host application.

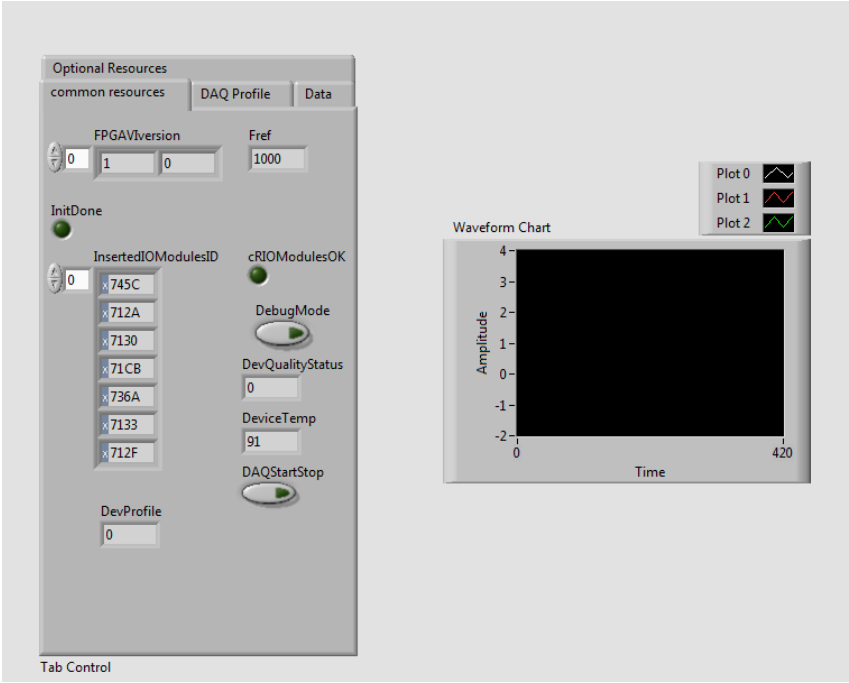

Fig. 34 Host HMI Application Front Panel

3.4.5 Point by Point DAQ Profile Example

3.4.5.1 *Objective*

The intention of the subsequent template is to implement the acquisition and generation of analog and digital signals using the cRIO platform. This template provides/generates one sample in every period of time configured in the cRIO (sampling rate).

3.4.5.2 *cRIO Hardware Elements Used*

Table 21 summarizes the cRIO modules used in the chassis and its allocation.

Table 21 Allocation of cRIO Modules in one NI9159 Chassis

NI9159 Slots							
1	2	3	4	5	6	7	8 - 14
NI9264	NI9205	NI9401	NI9477	NI9426	NI9476	NI9425	Free

3.4.5.3 *Signal Connection*

The signal connections for properly run this example are described in section 3.4.3.

3.4.5.4 *Mandatory Resources for Point by Point I/O Profile*

The mandatory resources for these types of systems are described in section 3.3.4.1.

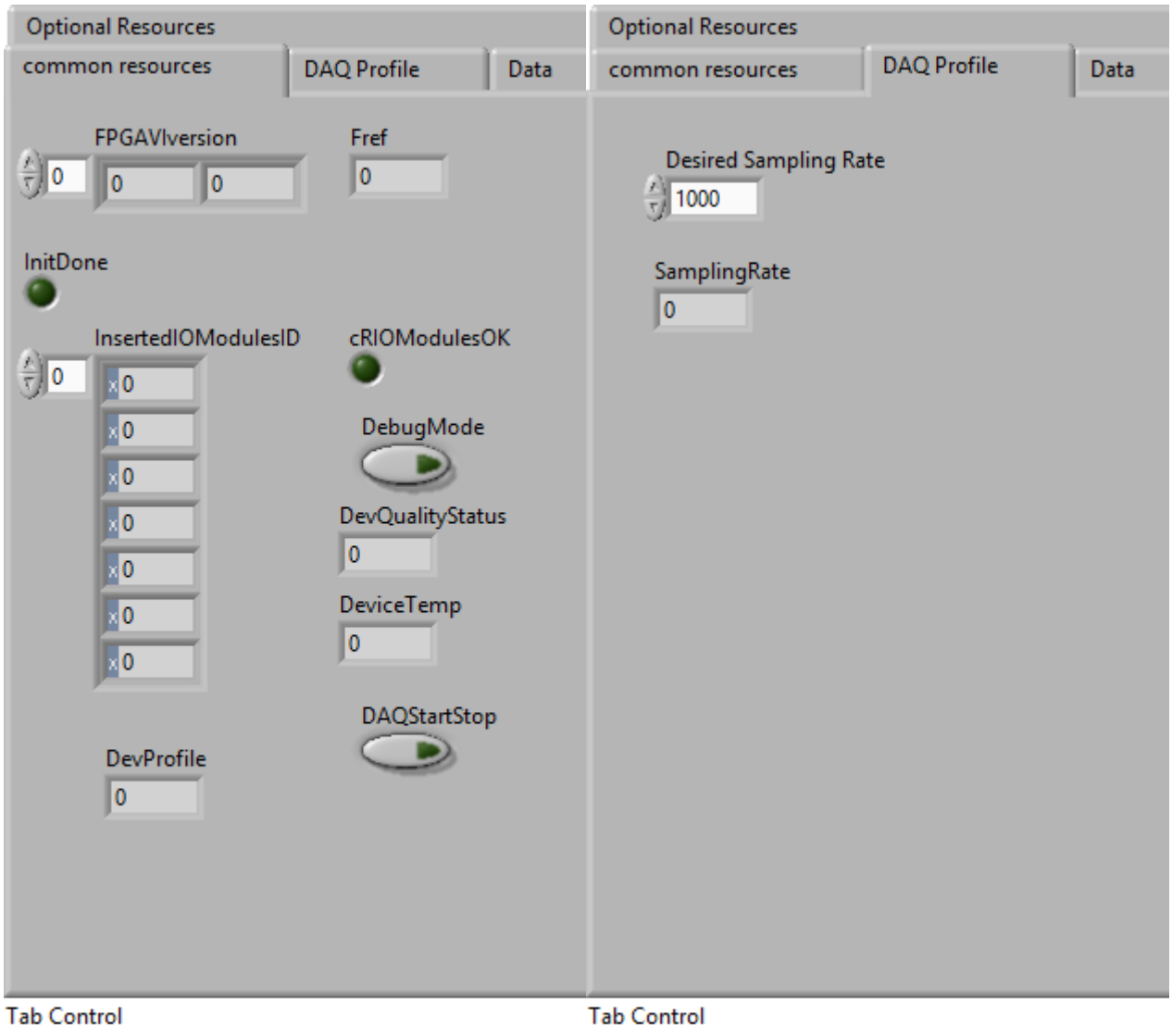

Fig. 35 Mandatory Resources for point by point I/O profile

3.4.5.5 *Optional Resources*

The optional resources of this type of profile are described in section 3.3.4.2. The optional resources implemented (see Fig. 36) in this example are used for:

- Analog Output signal generation
- Analog Input signal acquisition
- Digital Output signal generation
- Digital Input signal acquisition

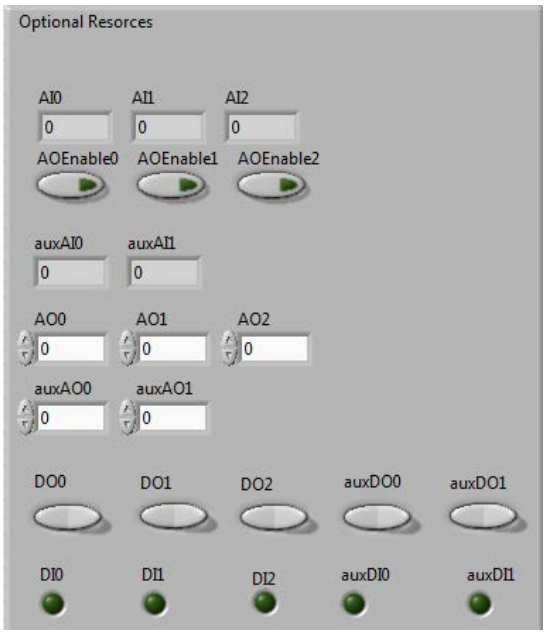

Fig. 36 Optional Resources Implemented in the Point by Point Example

3.4.5.6 *LabVIEW Implementation for a cRIO Point by Point DAQ*

Fig. 37 shows the LabVIEW project for this template, the FPGA target contains the instantiation of seven I/O adapter modules named Mod<n>, a FPGA implementation VI named coreIO.vi and the aforementioned host human machine interface files.

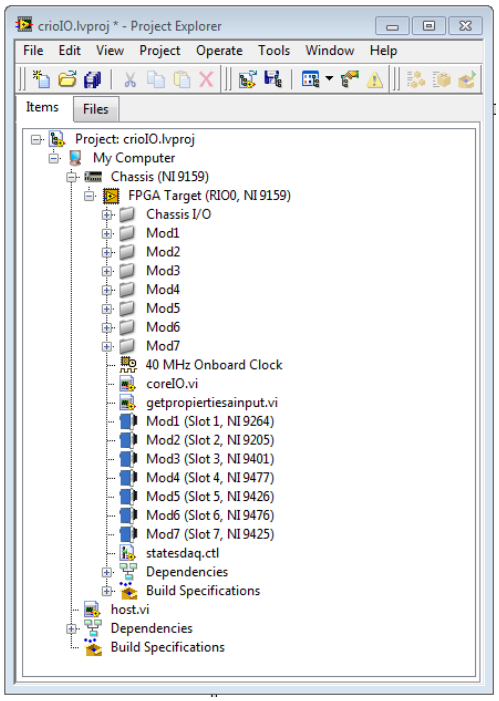

Fig. 37 LabVIEW Project for Point by Point Example

Before the implementation of the DAQ state machine the correct installation of the cRIO I/O adapter modules is verified as depicted in Fig. 38.

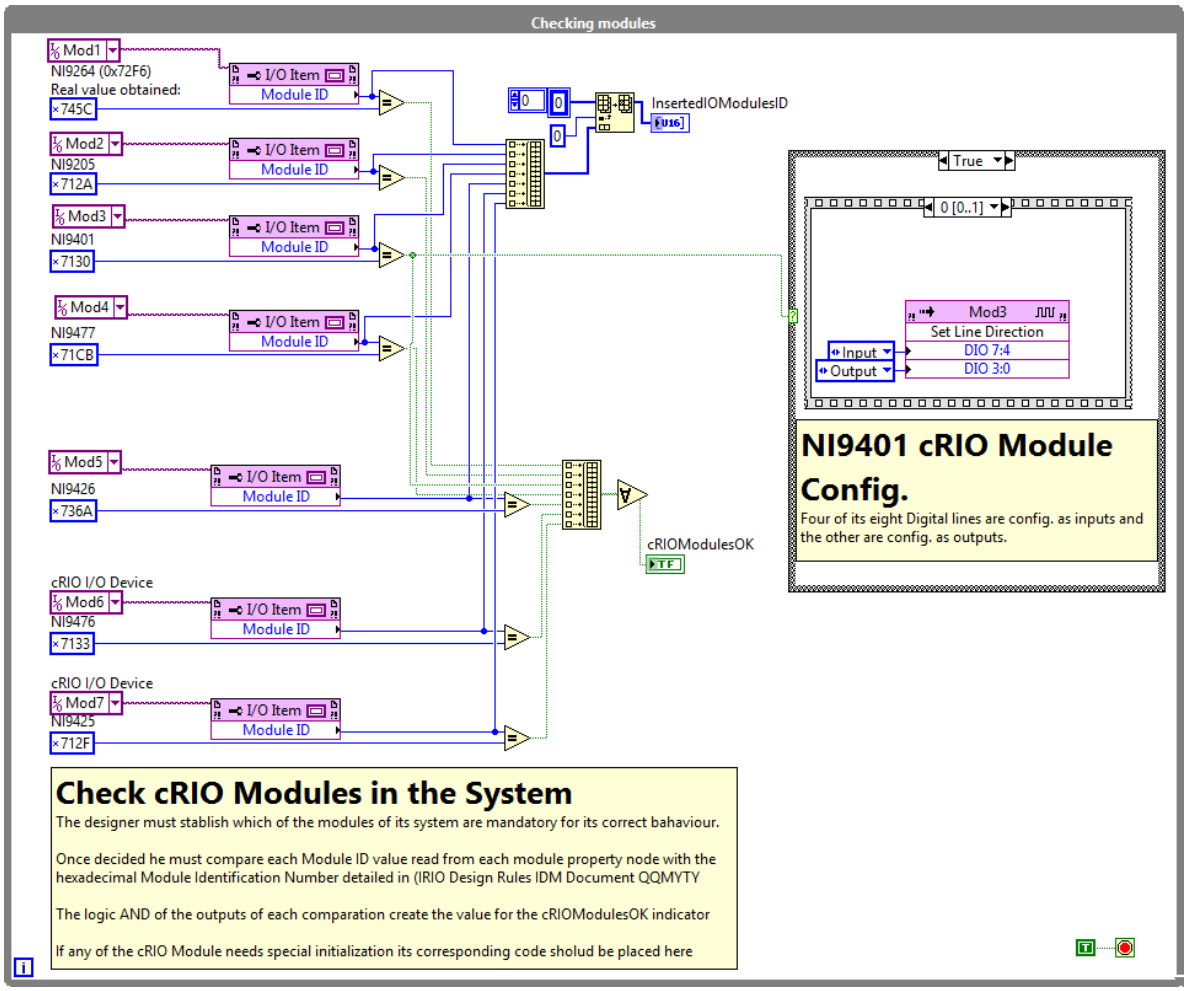

Fig. 38 Identification and Configuration of the Adapter Modules

Depicted in Fig. 39 the implementation of the acquisition and generation loop present the necessary resources to:

- Acquire from the NI-9205 analog input module
- Generation of analog output signals through the NI-9264 adapter module
- Generation of digital output values using NI-9401, NI-9477 and NI-9476
- Acquisition of digital values through NI-9401, NI-9426 and NI-9425 adapter modules
- Acquisition and generation of internal signals using the Aux<n> registers

Table 22 contains the detailed description of each of the controls and indicators present.

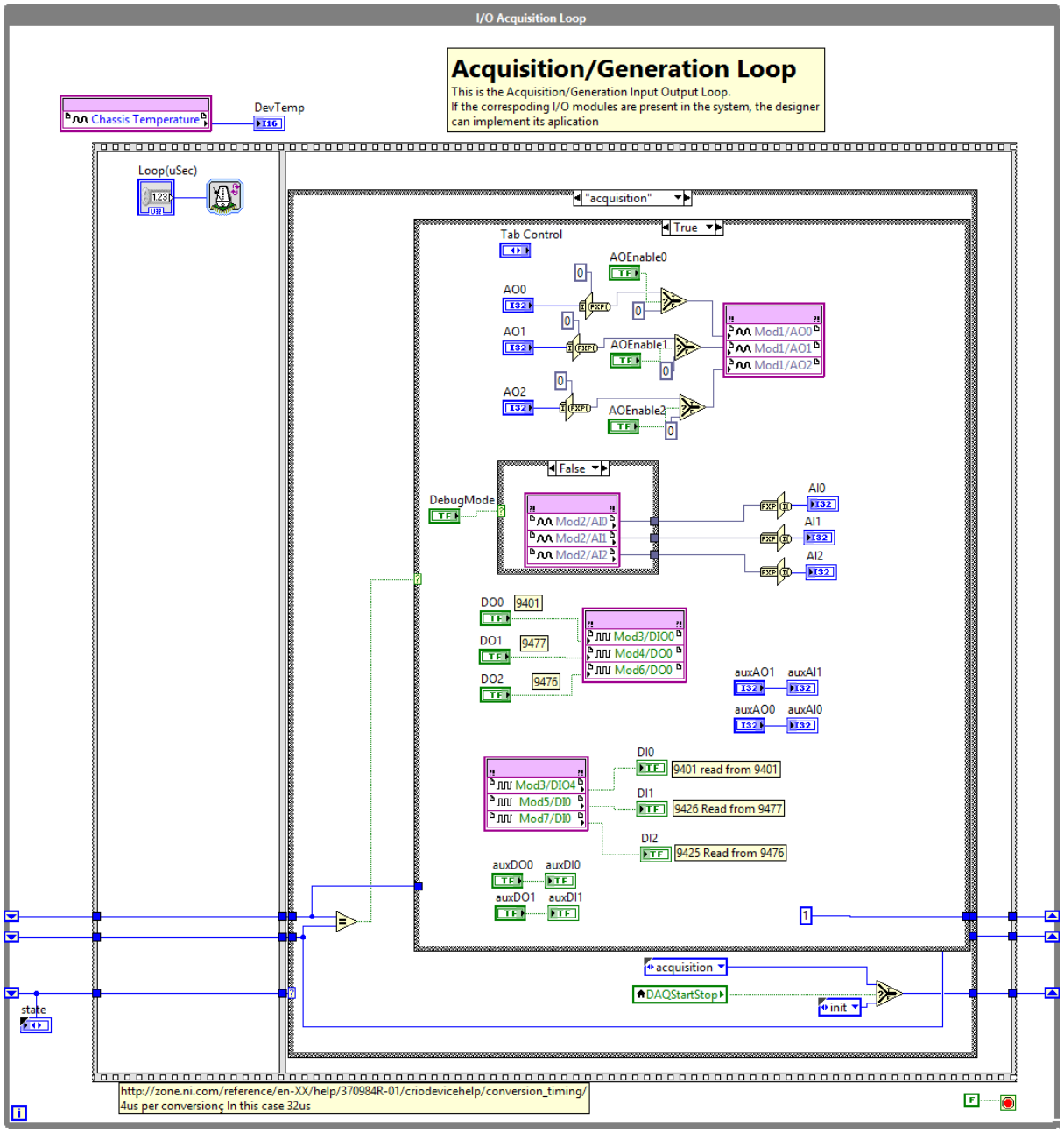

Fig. 39 I/O Acquisition Loop State Machine

Table 22: Resources identification

Terminals	I/O	Datatype	Info
AO<0-2>	Control	I32	If AOEnable<0-2> is true the I32 data converted to FXP is generated in its analog output pin of the NI9205 Module.

Terminals	I/O	Datatype	Info
AOEnable<0-2>	Control	Boolean	Enables the Analog Output signal generation for channel 0 to 2 respectively.
Debug Mode	Control	Boolean	If False acquisition from NI9205 is performed
AI<0-2>	Indicator	I32	NI9205 Acquisition and converted from FXP read to I32 data types
DO0	Control	Boolean	NI9401 Channel 0 generation
DO1	Control	Boolean	NI9477 Channel 0 generation
DO2	Control	Boolean	NI9476 Channel 0 generation
DI0	Indicator	Boolean	NI9401 Channel 4 read from NI9401 Channel 0
DI1	Indicator	Boolean	NI9426 Channel 0 read from NI9477 Channel 0
DI2	Indicator	Boolean	NI9425 Channel 0 read from NI9476 Channel 0
auxDO<0-1>	Control	Boolean	Terminals not connected to any output hardware, only connected internally to auxDI<0-1> respectively
auxDI<0-1>	Indicator	Boolean	Indicators wired to aforementioned auxDO<0-1>
auxAO<0-1>	Control	I32	Terminals not connected to any output hardware, only connected internally to auxAI<0-1> respectively
auxAI<0-1>	Indicator	I32	Indicators wired to aforementioned auxAO<0-1>
State	Indicator	Enum	Indicates the state of the I/O Acquisition Loop State Machine

3.4.5.7 *Host HMI Program*

The host program links to the FPGA controls and indicators in order to download the Bitfile to the target, configure its parameters and control and monitor the implemented instrument. The

input signals read are plotted in a waveform graph. Fig. 40 depicts the front panel of the HMI application, Fig. 41 depicts the structure of the HMI LabVIEW code, and Fig. 42 depicts the main acquisition loop of the host application which reads the AI<n>, AOEnable<n>, AO<n> and DebugMode registers from the FPGA.

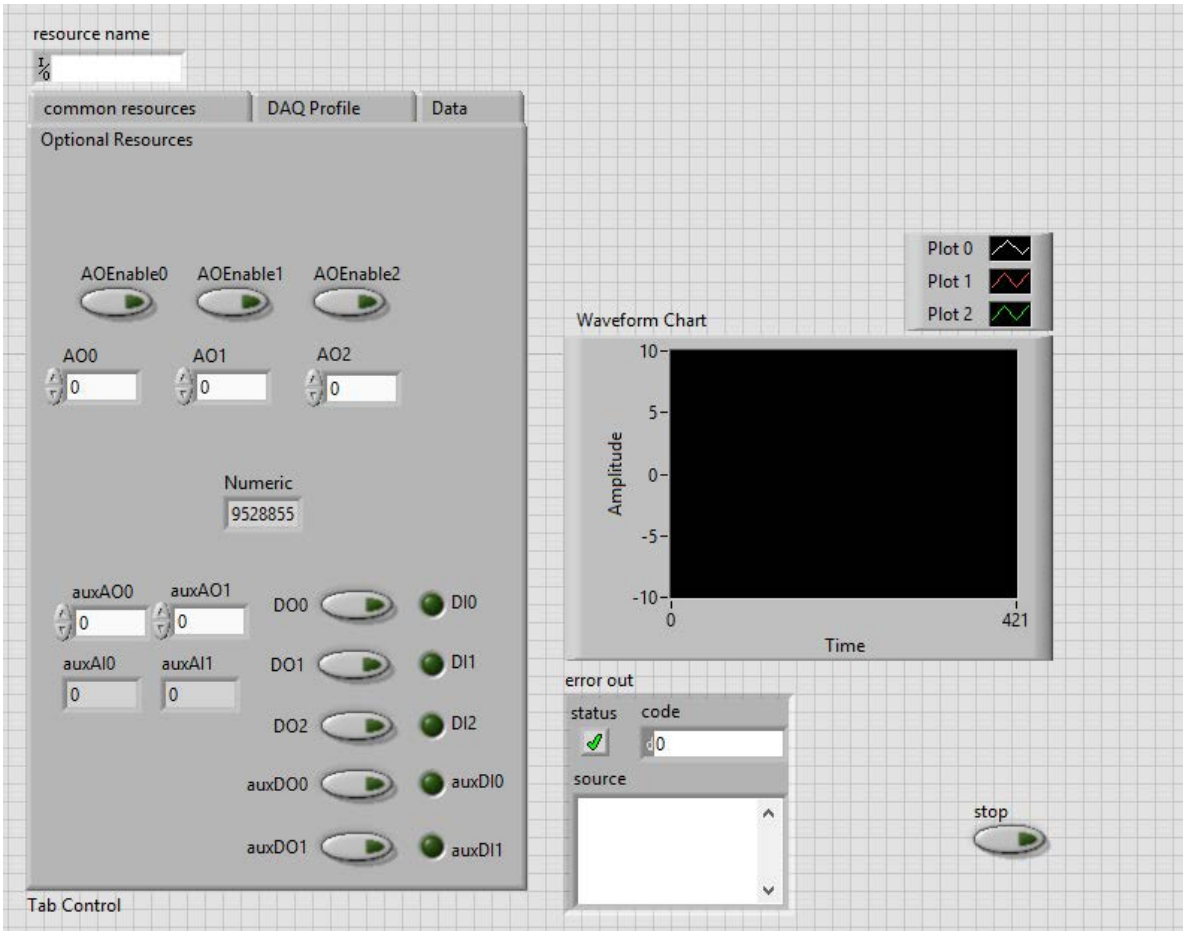

Fig. 40 HMI Front Panel of Point by Point Example

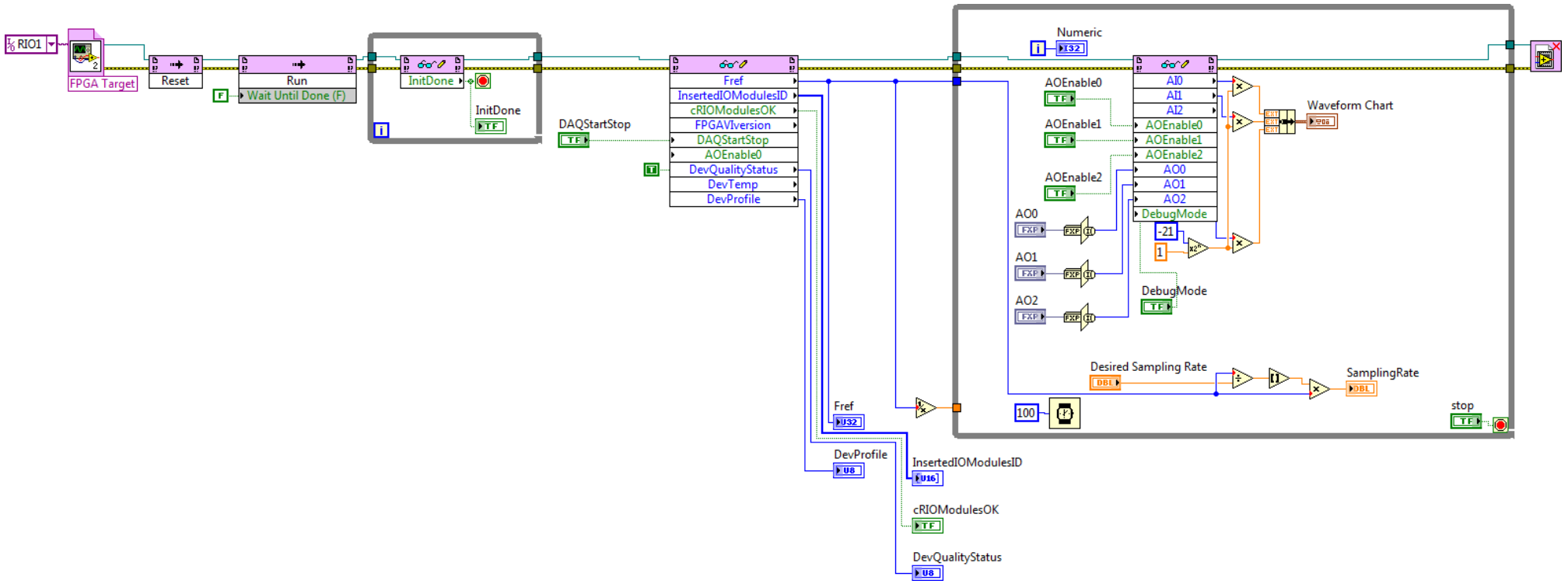

Fig. 41 Block Diagram of Point by Point Example

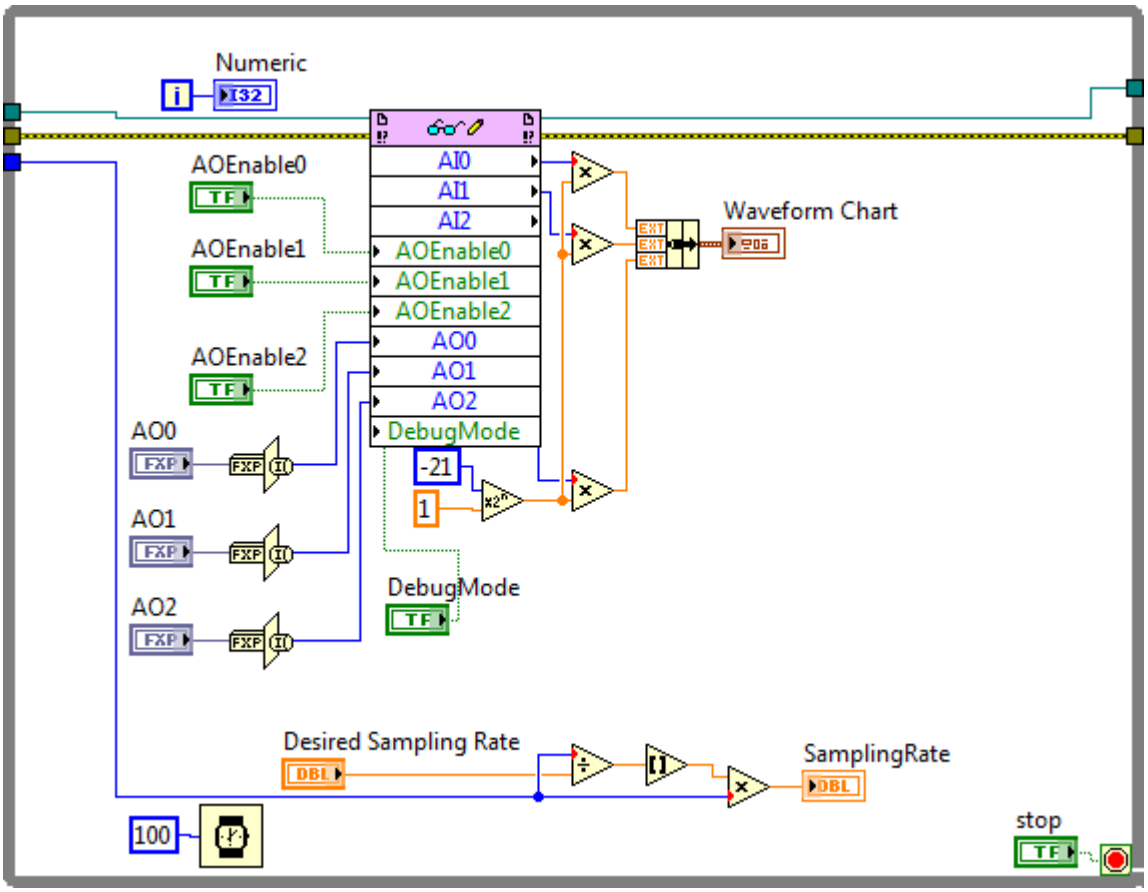

Fig. 42 Host Main Acquisition Loop of Point by Point Example

3.4.6 Analog Signal DAQ Profile (DMA based) Example

3.4.6.1 *Objective*

The intention of the subsequent template is to implement a use case of data acquisition applications acquiring continuously block of samples using DMA following the described design rules and name conventions.

3.4.6.2 *cRIO Hardware Elements Used*

Table 23 summarizes the cRIO modules used in the chassis and its allocation.

Table 23 Allocation of cRIO Modules in the NI9159 Chassis

NI9159 Slot							
1	2	3	4	5	6	7	8 - 14
NI9264	NI9205	NI9401	NI9477	NI9426	NI9476	NI9425	Free

NI9159 Slot							

3.4.6.3 *Signal Connection*

The signal connections for properly run this example are described in section 3.4.3.

3.4.6.4 *Mandatory Resources for Analog Signal DAQ Profile*

The mandatory resources for these type of systems is described in section 3.3.3.1

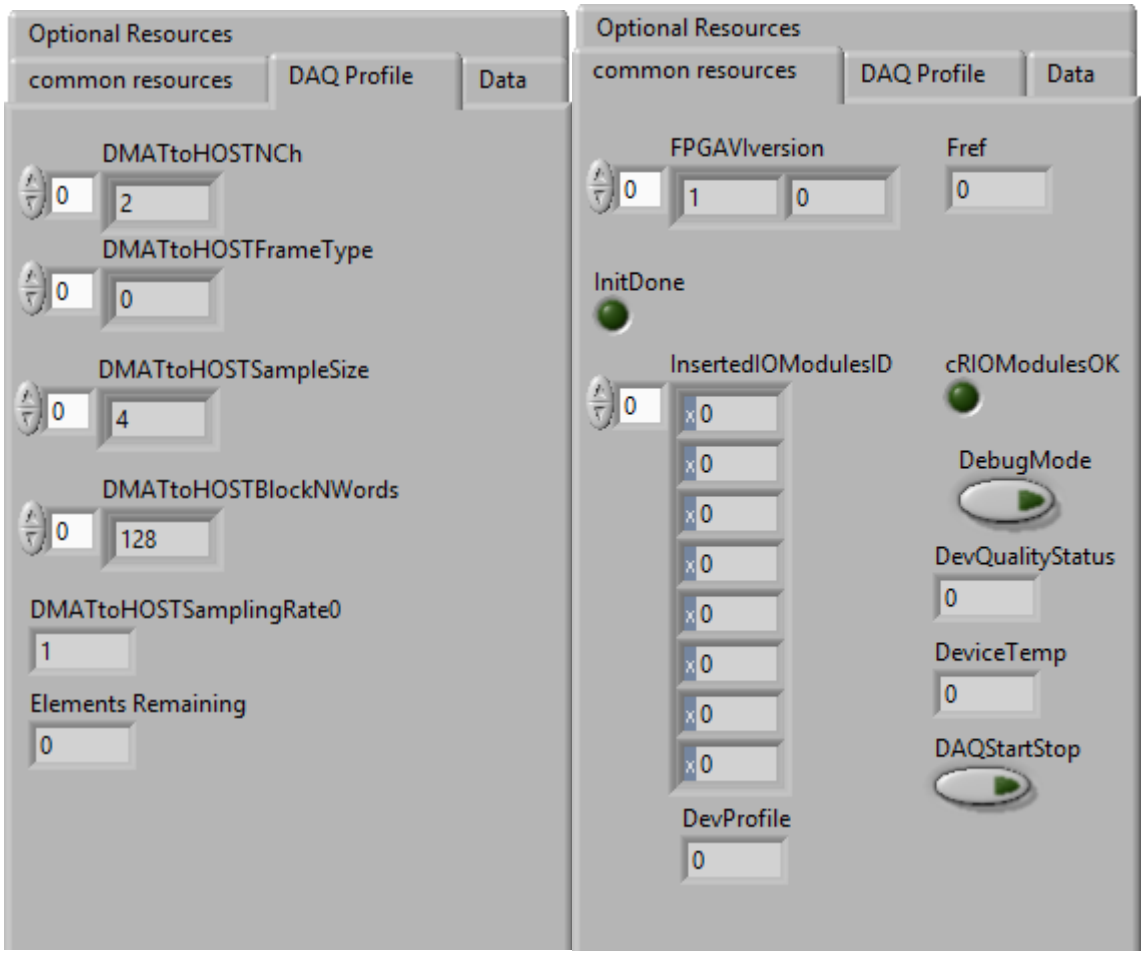

Fig. 43 Mandatory Resources for Analog Signal Example

3.4.6.5 *Optional Resources*

Optional resources for this type of profile are described in section 3.3.3.3

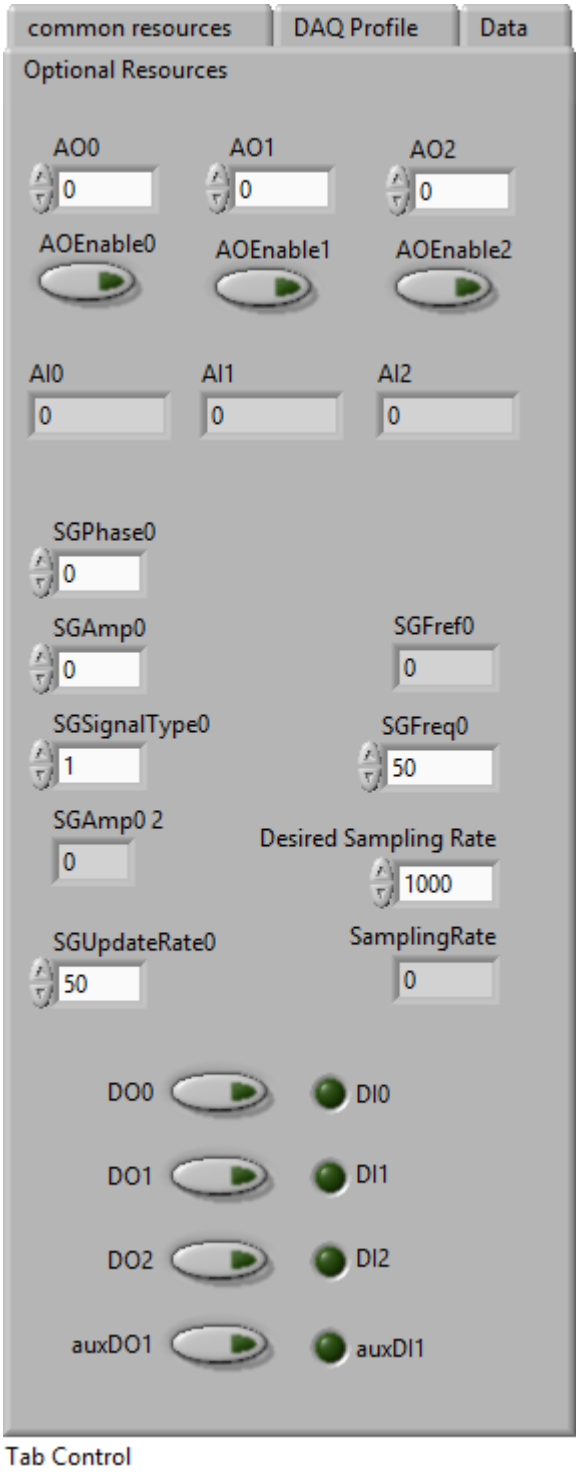

Fig. 44 Optional Resources for Analog Signal Example

3.4.6.6 *LabVIEW Implementation for a cRIO Analog Signal DAQ Profile*

Fig. 45 shows the LabVIEW project for this template. Fig. 46 depicts the acquisition loop of the DMA-based example for cRIO, the data acquired from channel AI0 and AI1 is packed and written to the DMA FIFO memory.

Fig. 47 depicts the implementation of the shape-selectable DDS signal generation loop. Table 24 describes in detail the functionality of the terminals depicted in aforementioned figures.

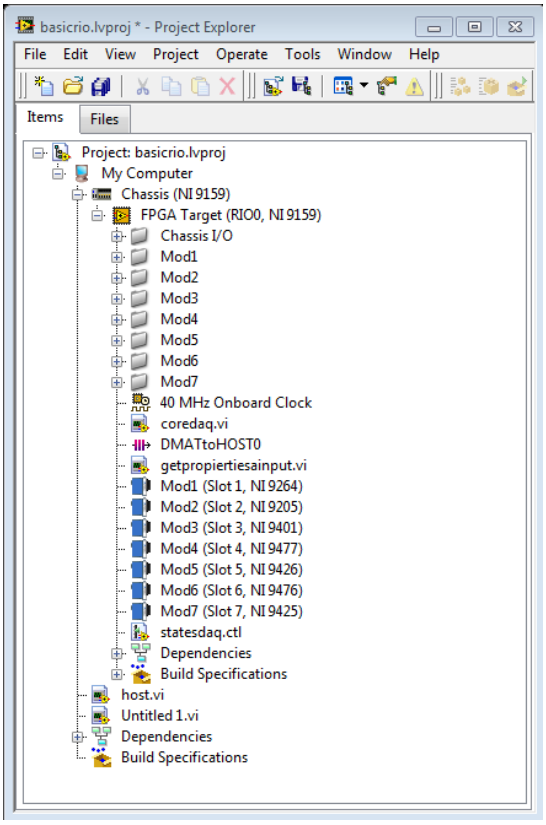

Fig. 45 LabVIEW Project for Analog Signal Example

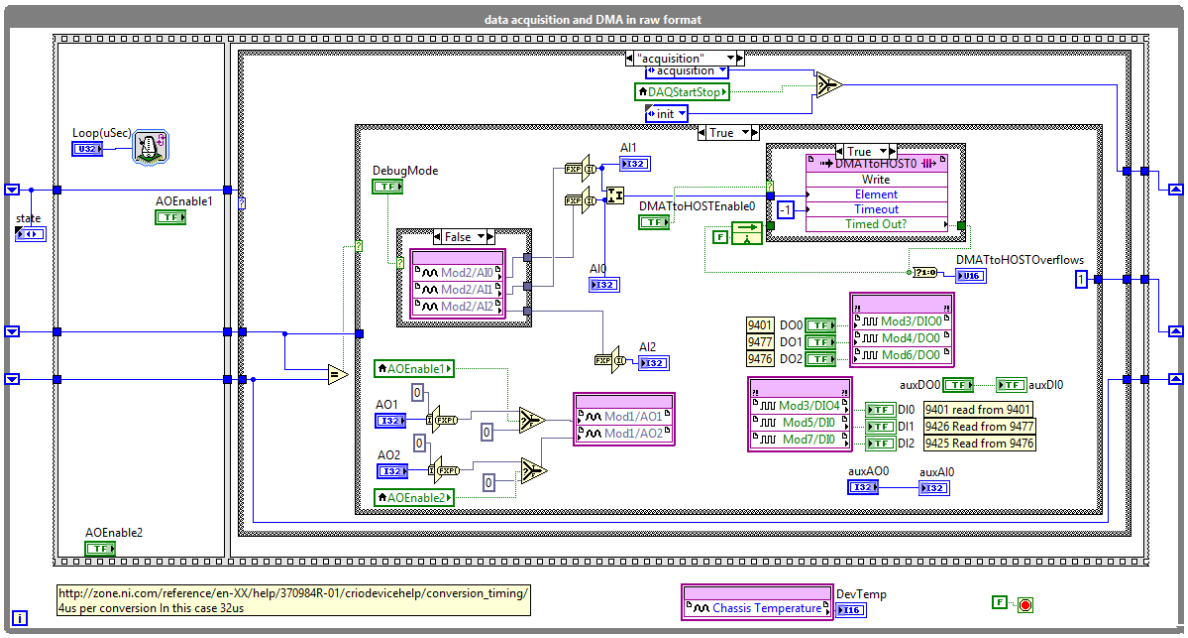

Fig. 46 Acquisition Loop State Machine

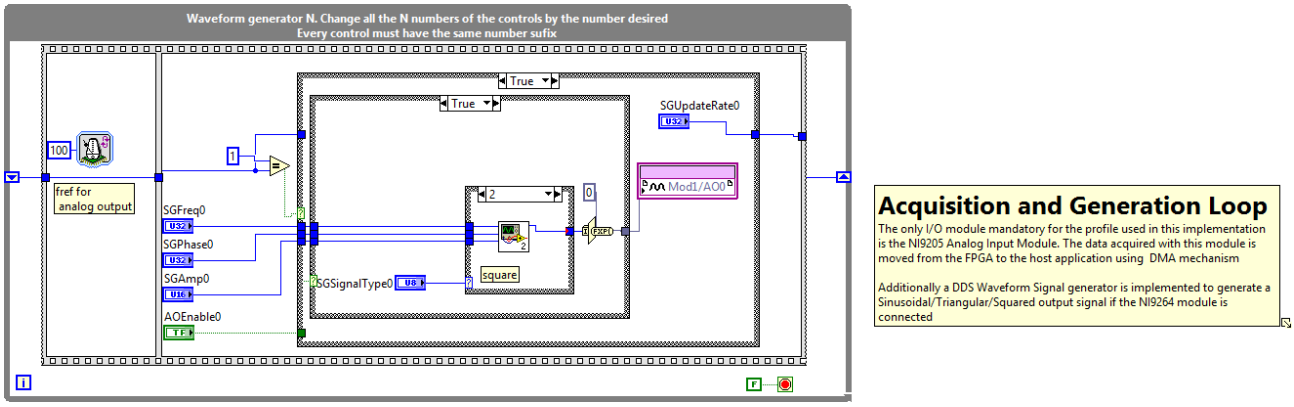

Fig. 47 DDS Signal Generation

Table 24: Summary of the terminals

Terminals	I/O	Datatype	Info
AO0	Control	I32	If AOEnable0 is TRUE the NI9264 output • AO0 (SGSignalType0 = 0) • DDS Sinusoidal Signal (SGSignalType0 = 1) • DDS Square Signal (SGSignalType0 = 2) • DDS Triangle Signal (SGSignalType0 = 3)
AO<1-2>	Control	I32	If AOEnable<1-2> is true the I32 data converted to FXP is generated in its analog output pin of the NI9205 Module.
AOEnable<0-2>	Control	Boolean	Enables the Analog Output signal generation for channel 0 to 2 respectively.
Debug Mode	Control	Boolean	If False acquisition from NI9205 is performed
AI<0-2>	Indicator	I32	NI9205 Acquisition and converted from FXP read to I32 data types
auxAO0	Control	I32	Terminal not connected to any output hardware, only connected internally to auxAI0 respectively
auxAI0	Indicator	I32	Indicator wired to aforementioned auxAO0
State	Indicator	Enum	Indicates the state of the I/O Acquisition

Terminals	I/O	Datatype	Info
			Loop State Machine
SGNo	Indicator	U8	See section 3.3.3.3
SGFreq0	Control	U32	See section 3.3.3.3
SGAmp0	Control	U16	See section 3.3.3.3
SGPhase0	Control	U32	See section 3.3.3.3
SGSignalType0	Control	U8	See section 3.3.3.3
SGUpdateRate0	Control	U32	See section 3.3.3.3

3.4.6.7 *Host HMI Program*

The host program links to the FPGA controls, indicators and DMAs in order to download the bitfile to the target, configure its parameters and control and monitor the implemented application. The DMA to Host is read in the host main acquisition loop (see Fig. 49 and Fig. 50) and the input signals are plotted in a waveform graph of the HMI LabVIEW front panel of Fig. 48.

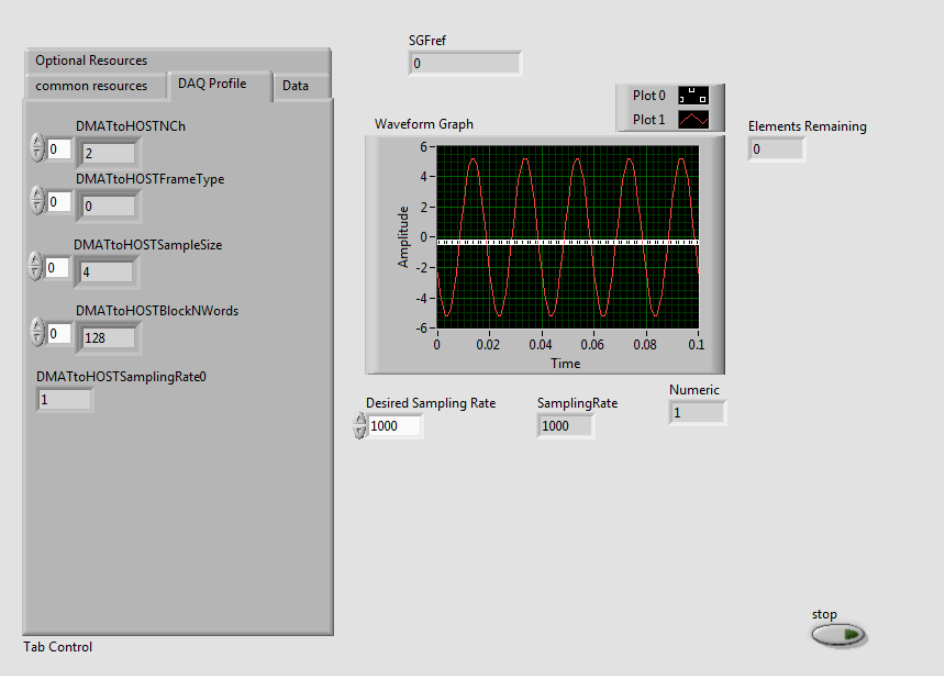

Fig. 48 HMI Front Panel of Analog Signal DAQ Example

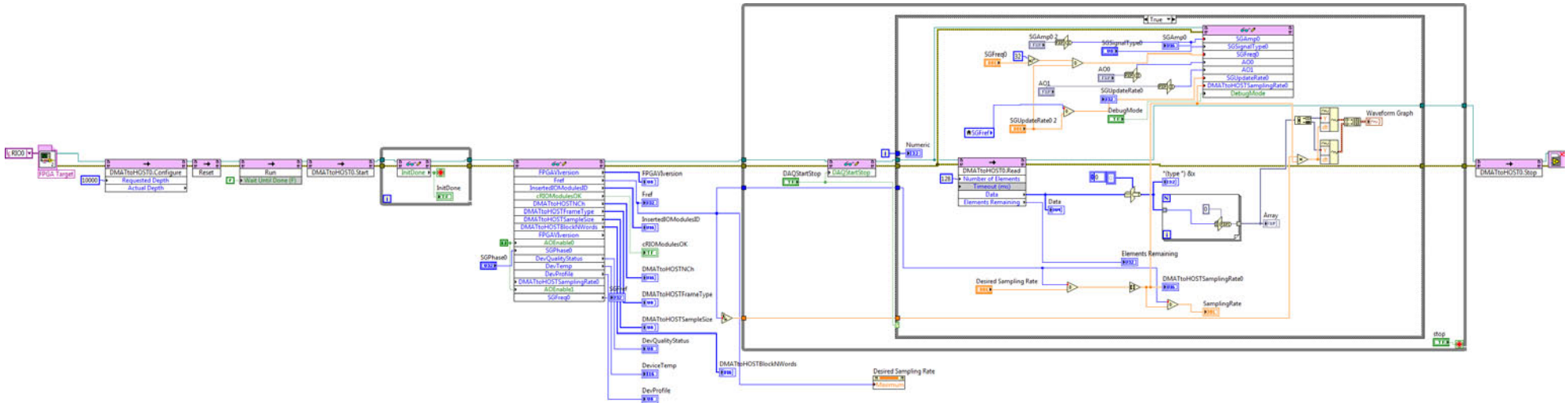

Fig. 49 Block Diagram of Analog Signal DAQ Example

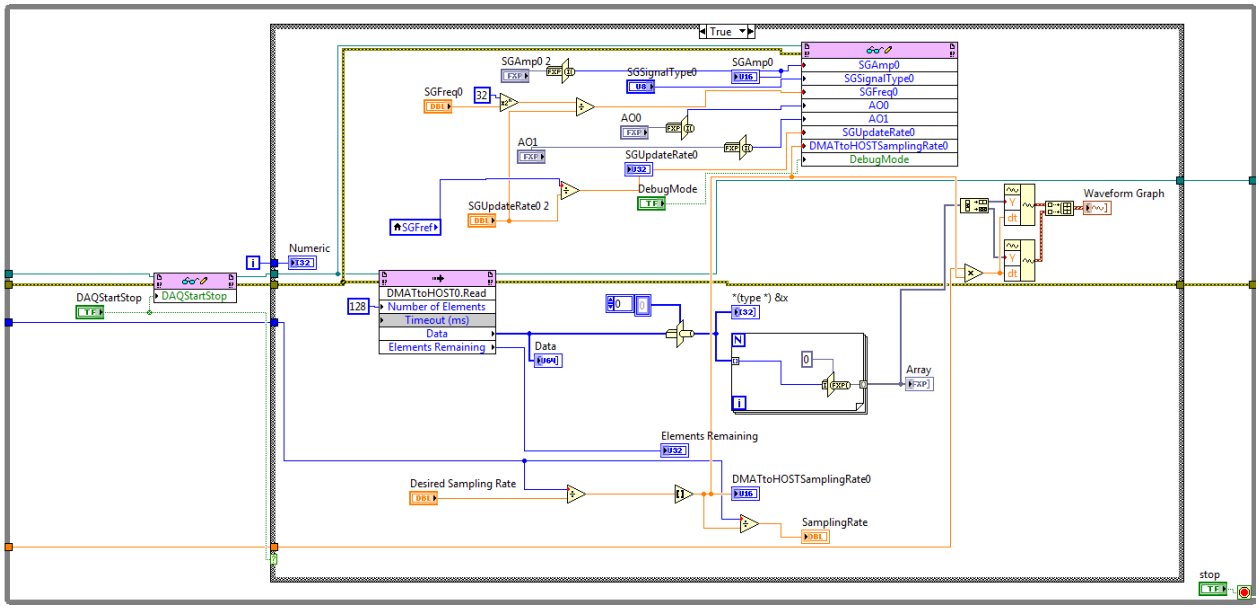

Fig. 50 Host Main Acquisition Loop of Analog Signal DAQ Example

3.5 NI FPGA Interface C API Generator

The FPGA Interface C API enables C/C++ applications to interact directly with compiled LabVIEW FPGA VIs on RIO devices without using LabVIEW. C/C++ applications can download a VI to a RIO target, perform DMA data transfers, wait on and acknowledge interrupts, and read from and write to named controls and indicators using C function calls.

A C/C++ application interfacing with the C API generated files interacts with VIs running on the FPGA of the RIO system. The C/C++ application can run on the real-time processor of a PXI system or the processor of a Windows or Linux PC, and interact with VIs running on the FPGA of a PXI or PCI RIO device.

3.5.1 Executing the Application

In the LabVIEW project window, the user must select the VI with the FPGA application and right click. In the pop-up menu, the user selects “Launch C API Generator”. A window immediately appears. The user must select the name of the bitfile with the design of the FPGA and an output directory for the headers file that will be generated with this application. Then, the user must click on the Generate button.

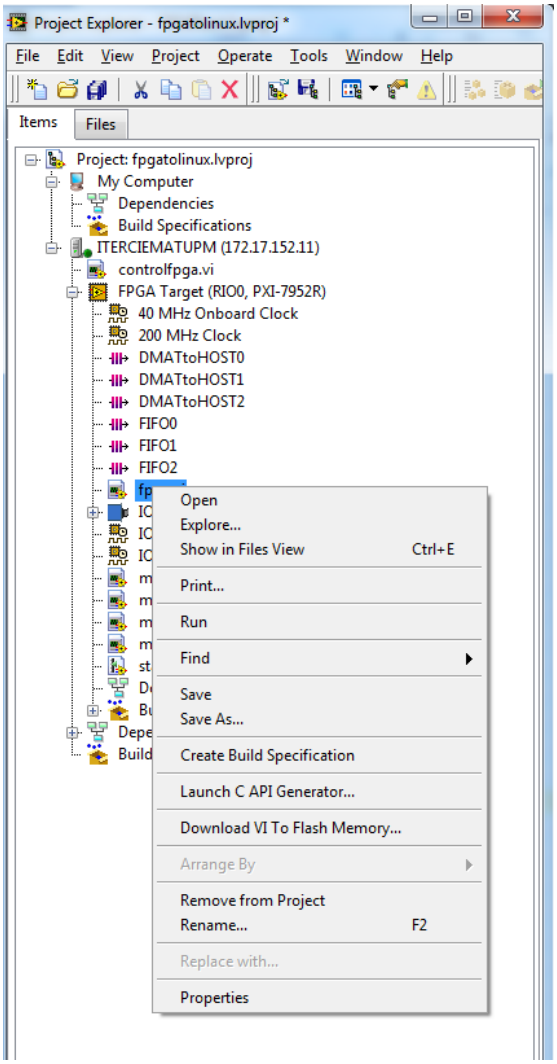

Fig. 51 Launching the C API Generator application

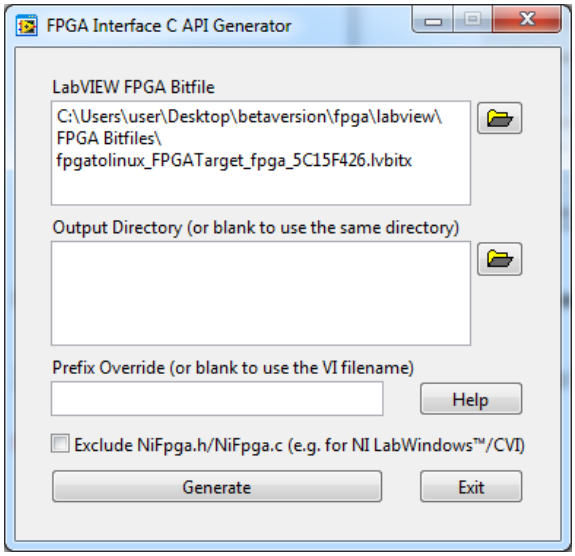

Fig. 52 Application C API generator in LabVIEW FPGA

C API generator program outputs four files, but only 2 are necessary.

- The first file is a renamed version of the bitfile with the name NiFpga_"*proyectName*".lvbitx.
- The second one is a header file with the name NiFpga_"proyectName".h, which includes the identification of the different control and indicators and the resources used in the FPGA design.

The header file contains many definitions with the different enumerated data types and their assignments, which will be used by the NI-RIO EPICS support driver to obtain the information for the FPGA.

3.5.2 Headerfile Generated

The NI FPGA INTERFACE C API Generator outputs a header file that will be used by the EPICS applications. To see the header files for the two data acquisition examples for cRIO please refer to section Appendix A.

4 IRIO LIBRARY OVERVIEW AND STAND-ALONE EXAMPLE APPLICATIONS

This chapter contains information about the library key concepts. It is important to understand how the FPGA resources can be used by other applications using this library.

The API provides the following functionalities:

- Download the bitfile to the FPGA
- Identification of the resources found in the FPGA if they are defined as described in the design rules.
- Profiles identification and data structure initialization according to these profiles.
- Setters and getters for performing input/output operations in the FPGA

4.1 FPGA Resources

The implementation of FPGA code has to meet the design rules. Once the design has been compiled and the bitfile have been tested using LabVIEW, the FPGA application is ready to be moved to CODAC Core System.

The header file and the bitfile obtained with C API generator have to be copied to /opt/codac/nirio folder in the subfolder “headerfile” and “bitfile”.

Depending on the profile implemented resources are organized in:

1. Common resources
2. Specific profile resources
3. Optional Resources.

The resources in the FPGA are identified using the specific labels described in [RD20].

4.2 IRIO Library Basic Use

The use of IRIO library requires following a sequence of specific steps. These steps are depicted in Fig. 53 and can be described as:

- Initialization of IRIO driver. This is accomplished with the *irio_initdriver* function. This function performs all the library initialization. The main operations implemented in the function are: download the bitfile to the FPGA, resources identification depending on the profile implemented in the FPGA and data structure initialization for later use by other API calls. This function returns a structure of type irioDrv_t that contains the mapping of all resources found in the FPGA.
- Use of getters and setters. The applications using the IRIO library can access to read/write the terminal available in the FPGA in order to know the initialization value or to set this. All getters/setters use a similar prototype with these parameters in common:
    - irioDrv_t* p_DrvPvt, this is data structure for the RIO device

    - TStatus* status, this is data structure containing the status and errors obtained once the function has been executed.
- If the application is using DMA (for image acquisition profiles or DAQ acquisition profile) the user needs to configure the DMA using irio_setupDMAstoHost.
- Execution of the hardware implemented in the FPGA. In the initialization the bitfile is downloaded but it is not executed. This means that the FPGA is waiting until the software send a run command. This run command is executed with the irio_setFPGAstart.
- Once the FPGA is running the application the user can interact with the FPGA terminals using the setters/getters and can trigger data acquisition retrieving the data using the irio_getDMATtoHost.
- For closing the driver and release the resources, irio_closedriver is required to be invoked.

Fig. 53 Basic steps to use IRIO library

4.3 Header Files of the IRIO Library

Table 25 enumerates the different header files required to use the IRIO library. Depending on the application one or more of the headerfiles have to be instantiated in the program.

Table 25 Header files for RIO library API

name	use
irioDataTypes.h	This header file contains all datatypes needed to use the API calls
irioDriver.h	This header file contains the basic functions for driver initialization and common resources access
irioHandlerAnalog.h	Prototypes for analog input management
irioHandlerDigital.h	Prototypes for digital input management
irioHandlerDMA.h	Prototypes for DMA data acquisition
irioHandlerImage.h	Prototypes for cameralink configuration and serial line use.
irioHandlerSG.h	Prototypes for signal generator management
irioResourceFinder.h	Prototypes for internal IRIO Library functions for FPGA resources managements

4.4 Stand-alone C Language cRIO Examples

Previous the integration with EPICS two stand-alone applications using the compactRIO LabVIEW templates have been created with the intention of debugging the IRIO library, creating easy-to-use examples to provide as part of the IRIO project development and create a step-by-step test for the hardware configured in the cRIO device. The examples provided for cRIO platform are numbered in Table 26.

Table 26 List of examples provided in the software unit for cRIO

Name	Utility	LabVIEW template used
cRIO_IO.c	Example of use of cRIO Point by Point profile in compactRIO platform	cRIO PBP
cRIO_DAQDMA.c	Example of use of cRIO DAQ profile in compactRIO platform	cRIO DAQDMA

The hardware configuration for these examples is the one described in Section 3.4 and the source code of the cRIO_IO.c and the cRIO_DAQDMA.c examples are provided in Appendix B.

4.4.1 Point by Point C Language Example Source Code Explanation

The initialization of the library and the IRIO device is done with the following call:

```
irio_initDriver("test",criomodel_serial,NIcriomodel,bitfileName,"V1.1",1
,bitFilePath,bitFilePath,&p_DrvPvt,&status );
```

This call downloads the bitfile in the FPGA, identifies all the resources implemented and returns the implemented resources data structure (p_DrvPvt) for later access to the FPGA.

The FPGA is set to *execution state* when irio_setFPGAStart with *value* parameter set to 1 call is performed. The execution of this function implies the detection of correct adapter module by the FPGA and the completion of the initialization steps defined by the hardware designer (LabVIEW for FPGA designer)

```
st|=irio_getFPGAStart(&p_DrvPvt, &aivalue,&status);
```

This call starts data acquisition in the RIO device.

```
st|=irio_setDAQStartStop(&p_DrvPvt,1,&status);
```

This call checks if data acquisition is started

```
st|=irio_getDAQStartStop(&p_DrvPvt,&aivalue,&status);
```

This function gets the VI version downloaded in the FPGA

```
st|=irio_getFPGAVIVersion(&p_DrvPvt,&value, 2, &valueLength,&status);
```

This function gets the temperature of the RIO device

```
st|=irio_getDevTemp(&p_DrvPvt,&aivalue,&status);
```

This function reads the profile implemented in the RIO device.

```
st|=irio_getDevProfifle(&p_DrvPvt,&aivalue,&status);
```

This function enables and disables the debug mode of the RIO FPGA.

```
st|=irio_setDebugMode(&p_DrvPvt,0,&status);
```

This segment of code reads the value of three analog inputs two times. The first one the debug mode is deactivates and in the second one is activated.

```
st|=irio_getAI(&p_DrvPvt,0,&aivalue,&status);
…
st|=irio_getAI(&p_DrvPvt,1,&aivalue,&status);

st|=irio_getAI(&p_DrvPvt,2,&aivalue,&status);

st|=irio_setDebugMode(&p_DrvPvt,1,&status);

st|=irio_getAI(&p_DrvPvt,0,&aivalue,&status);
…
st|=irio_getAI(&p_DrvPvt,1,&aivalue,&status);

st|=irio_getAI(&p_DrvPvt,2,&aivalue,&status);
```

This segment uses the analog input/output functions

```
st|=irio_setDebugMode(&p_DrvPvt,0,&status);
…
st|=irio_setAOEnable(&p_DrvPvt,0,NiFpga_True, &status);
…
st|=irio_getAOEnable(&p_DrvPvt,0,&aivalue,&status);
…
st|=irio_setAO(&p_DrvPvt,0,4000, &status);
…
st|=irio_getAO(&p_DrvPvt,0,&aivalue,&status);
…
st|=irio_getAI(&p_DrvPvt,0, &aivalue, &status);
…
st|=irio_setAO(&p_DrvPvt,1,8000, &status);
…
```

```
st|=irio_getAO(&p_DrvPvt,1,&aivalue,&status);
…
st|=irio_setAOEnable(&p_DrvPvt,1,NiFpga_True, &status);
…
st|=irio_getAOEnable(&p_DrvPvt,1,&aivalue,&status);
…
st|=irio_getAI(&p_DrvPvt,1, &aivalue, &status);
…
st|=irio_setAO(&p_DrvPvt,2,-8000, &status);
…
st|=irio_getAO(&p_DrvPvt,2,&aivalue,&status);
…
st|=irio_setAOEnable(&p_DrvPvt,2,NiFpga_True, &status);
…
st|=irio_getAOEnable(&p_DrvPvt,2,&aivalue,&status);
…
st|=irio_getAI(&p_DrvPvt,2, &aivalue, &status);
…
```

This other segment accesses the digital input/output

```
st|=irio_setDO(&p_DrvPvt,0,1, &status);
…
st|=irio_getDO(&p_DrvPvt,0,&aivalue, &status);
…
irio_getDI(&p_DrvPvt,0,&aivalue, &status);
…
st|=irio_setDO(&p_DrvPvt,0,0, &status);
…
st|=irio_getDO(&p_DrvPvt,0,&aivalue, &status);
…
st|=irio_getDI(&p_DrvPvt,0,&aivalue, &status);
…
st|=irio_setDO(&p_DrvPvt,1,1, &status);
…
st|=irio_getDO(&p_DrvPvt,1,&aivalue, &status);
…
st|=irio_getDI(&p_DrvPvt,1,&aivalue, &status);
…
st|=irio_setDO(&p_DrvPvt,1,0, &status);
…
st|=irio_getDO(&p_DrvPvt,1,&aivalue, &status);
…
st|=irio_getDI(&p_DrvPvt,1,&aivalue, &status);
…
st|=irio_setDO(&p_DrvPvt,2,1, &status);
…
irio_getDI(&p_DrvPvt,2,&aivalue, &status);
…
st|=irio_setDO(&p_DrvPvt,2,0, &status);
…
irio_getDI(&p_DrvPvt,2,&aivalue, &status);
…
st|=irio_setAuxAO(&p_DrvPvt,0,5000, &status);
```

```
…
st|=irio_getAuxAO(&p_DrvPvt,0,&aivalue,&status);
…
st|=irio_getAuxAI(&p_DrvPvt,0,&aivalue,&status);
…
st|=irio_setAuxAO(&p_DrvPvt,1,250, &status);
…
st|=irio_getAuxAO(&p_DrvPvt,1,&aivalue,&status);
…
st|=irio_getAuxAI(&p_DrvPvt,1,&aivalue,&status);
…
st|=irio_setAuxDO(&p_DrvPvt,0,1, &status);
…
st|=irio_getAuxDO(&p_DrvPvt,0,&aivalue,&status);
…
st|=irio_getAuxDI(&p_DrvPvt,0,&aivalue,&status);
…
st|=irio_setAuxDO(&p_DrvPvt,1,1, &status);
…
st|=irio_getAuxDO(&p_DrvPvt,1,&aivalue,&status);
…
st|=irio_getAuxDI(&p_DrvPvt,1,&aivalue,&status);
```

…

```
st|=irio_setDAQStartStop(&p_DrvPvt,0,&status);
```

```
st|=irio_closeDriver(&p_DrvPvt, &status);
```

4.4.2 DMA-Based C Language DAQ Example Source Code Explanation

To avoid unnecessary repetitions the IRIO library API functions described in this section are the one related to the DMA configuration and management because the rest of the functions used in this example are the same as described above.

The DMA functions used in this example are:

```
st|=irio_setUpDMAsTtoHost(&p_DrvPvt,&status);
st|=irio_cleanDMATtoHost(&p_DrvPvt,0,data, sizeof(data),&status);
st|=irio_setDMATtoHostSamplingRate(&p_DrvPvt,0,1, &status);
st|=irio_getDMATtoHostSamplingRate(&p_DrvPvt,0,&aivalues,&status);
st|=irio_getDMATtoHostData(&p_DrvPvt, 1, 0, data2, &elementsRead,
&status);
st|=irio_cleanDMAsTtoHost(&p_DrvPvt,&status);
st|=irio_closeDMAsTtoHost(&p_DrvPvt,&status);
```

Configure host memory and FPGA registers for DMA transfer. First, allocate host memory for DMA transfer and clean-up if possible trash data. Then set sampling rate, AD/DA conversion value, enable DMA channels, clean the DMA and close it.

Functions to configure the signal generator implemented in the FPGA:

```
st|=irio_setSGUpdateRate(&p_DrvPvt,0,10,&status);
st|=irio_getSGUpdateRate(&p_DrvPvt,0,&aivalues,&status);
st|=irio_setSGFreq(&p_DrvPvt,0,freq,&status);
st|=irio_getSGFreq(&p_DrvPvt,0,&aivalues,&status);
st|=irio_setSGSignalType(&p_DrvPvt,0,2,&status); //0=DC, 1=SINE 2=square
3=triangular
st|=irio_getSGSignalType(&p_DrvPvt,0,&aivalues,&status);
st|=irio_setSGAmp(&p_DrvPvt,0,Amp,&status);
st|=irio_getSGAmp(&p_DrvPvt,0,&aivalues,&status);
st|=irio_setSGPhase(&p_DrvPvt,0,0,&status);
st|=irio_getSGPhase(&p_DrvPvt,0,&aivalues,&status);
```

These methods configure the DDS signal generator connected to AO0 of the NI-9264 cRIO adapter module to generate a square signal with the amplitude determined by *Amp* and frequency *freq*.

5 EPICS & ASYNDRIVER

5.1 EPICS

5.1.1 Introduction to EPICS

EPICS is an open-source distributed control system toolkit that consists of a set of software tools and applications which provide a software infrastructure that application developers can use for building distributed control systems to operate devices such as Particle Accelerators, Large Experiments and major Telescopes. Such distributed control systems typically comprise a large amount of computers, networked together to allow communication among them and to provide control and feedback of the various parts of the device from a central control room, or even remotely over the internet EPICS uses a network-based client/server model. Large scale scientific applications often require hundreds of devices to communicate over a single network to form large distributed control systems. EPICS provides the standards and tools necessary to make this kind of communication possible.

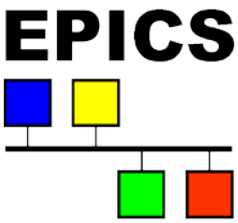

Fig. 54 EPICS Logo

Fig. 54 depicts the EPICS logo, each colored box represents a client or a server connected through a network. For EPICS, the ChannelAccess (CA) role can be Channel Access Client (CAC) or Channel Access Server (CAS), CACs are programs that require access to the Process Variables to carry out their purpose and the service provided by Channel Access Servers is the access to Process Variables.

The main components for EPICS are: the variables exchanged between clients and servers (Process Variables), the element that manipulates this variables (Input Output Controllers), backbone of the control network (CA protocol) and some tools used to monitor archive and edit these exchanged variables.

5.1.2 Process Variable

Process Variables (PVs), in the CODAC context often used in a narrower sense of EPICS PV, are EPICS variables that are exchanged between servers and clients and are defined by EPICS records in the EPICS Database. A process variable can give a computerized representation of a plant signal.

EPICS database is a process database running on a CAS, also known as the Input Output Controller (IOC); this database consists of records which represent data points of control system. Records consist of number of attributes (fields) and code that defines the records' behavior when active.

Most EPICS applications require only basic record types such as:

- Ai, ao: Analog input/output
- Bi, bo: Binary input/output
- Longin, longout: Long integer value input/output
- Mbbi, mbbo: Multi-bit binary input/output
- Stringin, stringout: String input/output
- Calc: Record that performs algebraic, relational and logical operation
- Waveform: Data in arrays.

Records can be connected among each other to exchange information and can be connected to hardware devices. Records can define database links to:

- Exchange data among each other (records connected among each other).
- Implement closed loop control.
- Records can connect to hardware devices and/or other records.

The IOC database manages Records. A Record has Fields, for instance a particular an analog input record can have fields such as

- VAL (value)
- EGU (Engineering Units)
- TIME (Timestamp)
- HOPR (High Operator Range)
- LOPR (Low Operator Range)
- STAT (Alarm Status)
- SEVR (Alarm Severity)

The database (.db) file can be used to get and set the contents of the fields of a record. These values and attributes of Process Variables are defined by the CA Servers. The Channel Access network protocol gives access to Channels. A particular Channel has properties such as: value, time stamp, units, upper control limit, lower control limit, status, and severity.

Main elements definition:

- A Process Variable: Typed structure according to a record type and the inputs, data manipulation and outputs are defined by EPICS records in the EPICS Database. These EPICS variables are exchanged between servers and clients. A PV is a typed structure according to a record type (like binary input, binary output, analog input, analog output, calculation ...) and the inputs, data manipulation and outputs are defined by configuring each record
- A Record Type: Predefined building block with a unique structure of fields and a unique processing routine to accomplish a specific function.
- Record support: Refers to a processing routine and the definition of the structure (i.e. an analog input record is used to monitor an analog signal, convert it to engineering

units, check the value against alarm limits, and notify interested channel access clients of any significant change).
- Record: A particular instance of a Record Type with appropriate values entered into the relevant fields.
- Database: A collection of records.
- Scanned (processed): Executing the record processing routine (unique to a record type) for a particular record

5.1.3 The Input Output Controller

The EPICS software processes are called IOCs. The main responsibility of the EPICS Input/Output Controller (IOC) is to input data from the local process (and/or the operator), manipulate/convert/compute it, update the PV value time-stamp and alarm status/severity and optionally output data to the local control process.

EPICS PVs become part of an IOC's database. The IOC scans the database, deciding when and how to process a predefined record. IOCs can run in the same environment as which it was compiled or can run in a different environment that where compiled using cross software development tools.

An IOC contains the following software components

- The IOC Database: The main element of an IOC is a database together with some structures that describe the contents of the database (the first field of a database record contains the record name).
- Database access routines via channel or database access routines.
- Mechanisms for deciding when to process a record (Scanners):
    - Periodic: To process a record periodically, standard scan rates are: 10, 5, 2, 1, 0.5, 0.2 and 0.1 seconds and custom scan rates can be configured up to speeds allowed by operating system and hardware.
    - Event driven: Events request from another record via links, EPICS Events and Channel Access Puts.
    - I/O Event: processing records based on external interrupts.
    - Passive: records are processed as a result of linked records being processed or as a result of external changes such as Channel Access puts.
    - Scan Once: Makes a record to be processed one time.
- Record support routines, device support routines and device drivers for accessing to external devices for each record. Record types not associated with hardware do not have device support or device drivers.
- The interface between the external world and the IOC via Channel Access.
- Database monitors provide a callback mechanism for database value changes. This allows the caller to be notified when database values change without constantly polling the database.
- Tools to implement state machines (Sequencer)

The IOC Core consists of the core software of EPICS that EPICS would not run without, that are: Channel Access, IOC Database, Scanners, Monitors, Database Definition Tools and Source/Release folders containing the raw code and the compiled code respectively.

5.1.4 The Channel Access

The Channel Access Protocol is a client-server TCP/IP-based communication protocol of EPICS. The protocol defines how Process Variable data is transferred between a server and client in any IOC database and also ensures an interface between the CODAC central control system and the local control systems. Each IOC provides a Channel Access Server which is prepared to establish communication with an arbitrary number of Channel Access Clients.

The main benefits of the CA are:

- Provides transparency from the Operating System
- Network transparency (equal access to remote and local channels)
- CPU architecture independence, isolation from software changes.

The software architecture paradigm is based on a publish/subscribe messaging throughout the control network. The requests are based on the PV name and that requests include Search, Get, Put and Add Event methods.

5.2 Device Support

Device support is the interface between record and the hardware; it hides hardware specific details from record processing routines. Device support routines are the interface between hardware specific fields in a database record and device drivers or the hardware itself.

Device support modules can be divided into two basic classes: synchronous and asynchronous. Synchronous device support is used for hardware that can be accessed without delays for I/O. Many register based devices are synchronous devices. Other devices, for example all General-Purpose Instrumentation Bus (GPIB) devices, can only be accessed via I/O requests that may take large amounts of time to complete. Such devices must have associated asynchronous device support. Asynchronous device support makes it more difficult to create databases that have linked records.

5.2.1 Overview of asynDriver

5.2.1.1 *Definitions*

asynDriver [RD21] is a software layer between device specific code and drivers that communicate with devices. It supports both blocking and non-blocking communication and can be used with both register and message based devices. asynDriver uses the following terminology:

- interface: All communication between software layers is done via interfaces. An interface definition is a C language structure consisting entirely of function pointers. An asynDriver interface is analogous to a C++ or Java pure virtual interface. Although the implementation is in C, the spirit is object oriented. Thus this document uses the term "method" rather than "function pointer".

- port A physical or logical entity which provides access to a device. A port provides access to one or more devices.
- portDriver Code that communicates with a port.
- portThread If a portDriver can block, a thread is created for each port, and all I/O to the portDriver is done via this thread.
- device A device (instrument) connected to a port. For example a GPIB interface can have up to 15 devices connected to it. Other ports, e.g. RS-232 serial ports, only support a single device. Whenever this document uses the word device without a qualifier, it means something that is connected to a port.
- device support Code that interacts with a device.
- synchronous Support that does not voluntarily give up control of the CPU.
- asynchronous Support that is not synchronous. Some examples of asynchronous operations are epicsThreadSleep, epicsEventWait, and stdio operations. Calls to epicsMutexTake are considered to be synchronous operations, i.e. they are permitted in synchronous support.
- asynDriver The name for the support described in this manual. It is also the name of the header file that describes the core interfaces.
- asynManager An interface and the code which implements the methods for interfaces asynManager and asynTrace.
- asynchronous Driver A driver that blocks while communicating with a device. Typical examples are serial, gpib, and network based drivers.
- synchronous Driver A driver that does not block while communicating with a device. Typical examples are VME register based devices.
- Message Based Interfaces that use octet arrays for read/write operations.
- Register Based Interfaces: Interfaces that use integers or floats for read/write operations.
- Interrupts, implemented by asynManager.

Standard interfaces are defined so that device specific code can communicate with multiple port drivers. For example if device support does all its communication via reads and writes consisting of 8 bit bytes (octets), then it should work with all port drivers that support octet messages. If device support requires more complicated support, then the types of ports will be more limited. Standard interfaces are also defined for drivers that accept 32 bit integers or 64 bit floats. Additional interfaces can be defined, and it is expected that additional standard interfaces will be defined.

Multiple layers can exist between device specific code and a port driver. For more complicated protocols, additional layers can be created.

A driver normally implements multiple interfaces. For example asynGpib implements asynCommon, asynOctet, and asynGpib.

asynManager uses the Operating System Independent features of EPICS base. It is, however, independent of record/device support. Thus, it can be used by other code program.

5.2.1.2 *Standard Interfaces*

These are interfaces provided by asynManager or interfaces implemented by all or most port drivers. The interfaces are:

- asynManager provides services for communicating with a device connected to a port.
- asynCommon is an interface that must be implemented by all low level drivers. The methods are:
  - report - Report status of port.
  - connect - Connect to the port or device.
  - disconnect - Disconnect from the port or device.
- asynTrace is an interface for generating diagnostic messages.
- asynLockPortNotify is an interface that is implemented by a driver which is an asynUser of another driver.
- asynDrvUser is an interface for communicating information from device support to a driver without the device support knowing any details about what is passed.

5.2.1.3 *Generic Interfaces*

In addition to asynCommon and optionally asynDrvUser, port drivers can implement one or more of the following message and/or register based interfaces.

- asynOctet methods for message based devices
- asynFloat64 methods for devices that read/write IEEE float values
- asynFloat32Array methods for devices that read/write arrays of IEEE 32-bit float values
- asynFloat64Array methods for devices that read/write arrays of IEEE 64-bit float values
- asynInt32 methods for devices that read/write integer values. Many analog I/O drivers can use this interface.
- asynInt8Array methods for devices that read/write arrays of 8-bit integer values
- asynInt16Array methods for devices that read/write arrays of 16-bit integer values
- asynInt32Array methods for devices that read/write arrays of 32-bit integer values
- asynUInt32Digital methods for devices that read/write arrays of digital values. This interface provides a mask to address individual bits within registers.
- asynGenericPointer methods for devices that read/write arbitrary structures, passed via a void* pointer.

6 IRIO ASYN FUNCTIONALITY

This chapter describes the implemented support for RIO devices [RD22], in concrete to cRIO devices included as part of the IRIO project for exporting the hardware functionalities implemented in FPGA to EPICS. Thus implementation of asynDriver interfaces and reasons with its associated getters and setters allows interfacing with the IRIO library to access to the FPGA populated registers.

6.1 Device Support Functions

The NI-RIO EPICS driver is provided with functions available from the st.cmd file or from the EPICS shell command line. All functions are listed and described here are used for configuring various device parameters.

6.1.1 Device Functions

6.1.1.1 *nirioinit*

```
int nirioinit(const char *namePort,const char *DevSerial,const char
*PXInirioModel, const char *projectName, const char *FPGAversion,int
verbosity)
```

Initialize the NI-RIO EPICS device support. Use this function to initialize the NI-RIO EPICS device support. This function registers all needed functionality of the ASYN driver used in this device support and registers the port name provided with the function call. This function has to be added in the usrPreDriverConf.cmd file, which executes before the iocInit function in st.cmd file. Once the function is executed the NI-RIO EPICS driver is ready to be used and records can connect using asynDriver interfaces. The description of the *nirioinit* function parameters are described in Table 27.

Table 27 nirioinit function parameters description

Parameter Name	Description.	Example
namePort	Port name to associate with the selected device.	RIO_0
DevSerial	Serial number of the RIO device.	ffbbeecc
PXInirioModel	RIO device model.	PXIe7966R
projectName	Name identifying the design to load in the FPGA.	LabVIEW-VI

Parameter Name	Description.	Example
FPGAVersion	Version of the bitfile used.	V1.0
verbosity	Parameter for displaying internal driver messages for debugging purposes.	1

6.2 Supported Records

This section describes records used by NI-RIO EPICS driver. The records to be used depend on the implementation done in the FPGA. Assuming that RIO's FPGA contain a valid design the records can connect to the different asynDriver interfaces for performing operations. The number of records and the type of these records are related with the profiles implemented in the FPGA. Check this information provided by the designer of the FPGA code to understand which PVs can be used. Process Variable (PV) names follow I&C Signal Process Variable Naming Convention. This section assumes that the reader is familiar with EPICS and that CODAC Core System (version 5.1 or later) and the ASYN driver is installed. The NI-RIO EPICS driver is implemented as an Asyn port driver. AsynDriver provides generic device support for standard EPICS records [RD21]. The Asyn generic support uses the following conventions for DTYP and INP. OUT fields are the same as INP.

```
field(DTYP,"asynXXX")
field(INP,"@asyn(portName,addr,timeout) drvParams")
```

Where:

- XXX-Supported interface type name.
- portName –Name of the port.
- Addr – Address of the port. If addr is not specified the default is 0.
- timeout - Timeout value for asynUser.timeout. If not specified the default is 1.0.
- drvParams – Optional value passed to the low level driver via the asynDrvUser interface.

For example:

```
record(ai, "$(PORT_NAME):AI$(ADDR ") {
field(SCAN, "1 second")
field(DTYP, "asynInt32")
field(INP, "@asyn($(PORT_NAME),$(ADDR))AI")
}
```

The NI-RIO EPICS driver implements the common, asynDrvUsr, asynOctet, asynInt8Array, asynInt32, asynInt32Array, asynFloat64 and asynFloat32Array interfaces. The driver uses

address to select the channel when applicable. The interface parameters are described in subsequent sections.

6.3 NI-RIO Device Templates

Table 28 summarizes the templates available to be used with RIO devices. Some templates are common to both platforms: cRIO and FlexRIO. In that case the identification of the template starts with *rio*. Depending on the profile implemented in the RIO's FPGA the user have to instantiate the templates listed in Table 28. M means mandatory and O is optional. For instance, for a cRIO implementation of the Point by Point profile the user has to instantiate mandatory the templates: rio-module.template, rio-pointbypoint.template. The remaining templates are optional depending on the final application implemented in the cRIO system. For instance, if the user is using a NI9205 (Analog Input module with 32 channels) users can instantiate this template and use up to 32 input channels (if implemented in the FPGA). Every channel will be identified using the address.

Table 28: Template identification

Target	Profile	Template	Name	Mandatory Optional
cRIO	**Point by Point Profile.**	Common resources	rio-module.template	M
		Point by Point template	rio-pointbypoint.template	M
		Analog Input	rio-ai.template	O
		Aux Analog Input	rio-auxai.template	O
		Analog Output	rio-ao.template	O
		Aux Analog Output	rio-auxao.template	O
		Digital Input	rio-di.template	O
		Aux Digital Input	rio-auxdi.template	O
		Digital Output	rio-do.template	O
		Aux Digital Output	rio-auxdo.template	O

Target	Profile	Template	Name	Mandatory Optional
		Signal Generation	rio-sg.template	O
cRIO	**Data acquisition Profile.**	Common resources	rio-module.template	M
		Data Acquisition	rio-dataacquisition.template	M
		Waveforms	rio-wf.template	M
		Analog Input	rio-ai.template	O
		Aux Analog Input	rio-auxai.template	O
		Analog Output	rio-ao.template	O
		Aux Analog Output	rio-auxao.template	O
		Digital Input	rio-di.template	O
		Aux Digital Input	rio-auxdi.template	O
		Digital Output	rio-do.template	O
		Aux Digital Output	rio-auxdo.template	O
		Signal Generation	rio-sg.template	O

This section provides details of the records used in the templates which are part of the NI-RIO EPICS Device Driver (version 1.1.0 or higher). In the substitution files, the following macros are defined:

- CBS1: Control break down structure level 1 name
- CBS2: Control break down structure level 2 name
- BOARDTYPEIDX: Controller index
- MODULEIDX: Module index
  - PORT name will be RIO_$(MODULEIDX), i.e. RIO_0 for module instance 0.
- CHIDX : Channel number
- GRIDX: Group number
- NSAMPLES: Number of samples in waveforms.

6.3.1 Interfaces and Reasons

The asynDriver uses the following interfaces and reasons:

Table 29 AsynDriver Interfaces and Reasons Created

Interface	Reason	Address
OctetRead	VIversion	N/A (0)
	FPGAStatus	N/A (0)
	InfoStatus	N/A (0)
	UARTReceive	Group of Channels
OctetWrite	UARTTransmit	Group of Channels
Int32Read/Write	FPGAStart	N/A (0)
	DAQStartStop	N/A (0)
	debug	N/A (0)
	DevQualityStatus	N/A (0)
	AOEnable	Channel
	DI	Channel
	DO	Channel
	auxAI	Channel
	auxAO	Channel
	auxDI	Channel
	auxDO	Channel
	GroupEnable	Group of Channels
	SamplingRate	Group of Channels
	DMAsOverflow	Group of Channels
	DF	Group of Channels

Interface	Reason	Address
	SGUpdateRate	Channel
	SGFreq	Channel
	SGPhase	Channel
	CLSignalMapping	Group of Channels
	CLConfiguration	Group of Channels
	CLLineScan	Group of Channels
	CLFVALHigh	Group of Channels
	CLLVALHigh	Group of Channels
	CLDVALHigh	Group of Channels
	CLSpareHigh	Group of Channels
	UARTBreakIndicator	Group of Channels
	UARTFrammingError	Group of Channels
	UARTOverrunError	Group of Channels
	CLSizeX	Group of Channels
	CLSizeY	Group of Channels
	SGSignalType	Channel
Float64Write	AO	Channel
	SGAmp	Channel
Float64Read	DeviceTemp	Channel
	AI	Channel
	AO	Channel
	SGAmp	Channel
Float32ArrayRead	CH	Channel

Interface	Reason	Address
Int8ArrayRead	CH	Channel

6.4 Records

6.4.1 Common Resources Records for all RIO Devices

This paragraph describes all mandatory PVs used to manage common resources in cRIO. These PVs are included in rio-module.template installed with irio module in */opt/codac/epics/modules/irio/db*.

6.4.1.1 *Run the FPGA VI*

Use this record to run the VI in the FPGA. A TRUE value starts the execution of the hardware implemented in the FPGA.

```
record(bo,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-
FPGASTART"){
     field(DESC,"FPGA Start.Does not stop FPGA.")
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),0)FPGAStart")
     field(PINI,"NO")
     field(ZNAM,"OFF")
     field(ONAM,"ON")
}
```

6.4.1.2 *FPGA VI version*

This record allows retrieving the version of the VI implemented in the FPGA. The VI version contains a mayor and a minor value.

```
record(stringin,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-
FPGAVIVERSION"){
     field(DESC,"Show the bifile version")
     field(DTYP,"asynOctetRead")
     field(INP,"@asyn(RIO_$(MODULEIDX),0)VIversion")
     field(SCAN,"1 second")
     field(PINI,"NO")
}
```

6.4.1.3 *RIO Device temperature*

This record allows reading the temperature of the RIO device.

```
record(ai,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-
DEVICETEMP"){
     field(DESC,"FPGA Temperature")
     field(DTYP,"asynFloat64")
     field(INP,"@asyn(RIO_$(MODULEIDX),0)DeviceTemp")
     field(SCAN,"1 second")
```

```
    field(PREC,"6")
    field(PINI,"NO")
    field(LINR,"NO CONVERSION")
    field(EGU,"C Degrees")
}
```

6.4.1.4 *Data acquisition start/stop*

This record starts/stop the data acquisition process in the RIO device.

```
record(bo,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-
DAQSTARTSTOP"){
    field(DESC,"Controls DAQ Start/Stop ")
    field(DTYP,"asynInt32")
    field(OUT,"@asyn(RIO_$(MODULEIDX),0)DAQStartStop")
    field(PINI,"NO")
    field(ZNAM,"OFF")
    field(ONAM,"ON")
}
```

6.4.1.5 *FPGA Debug Mode*

This record sets the FPGA in debug mode. The behaviour of debug mode is defined by the hardware developer. Check the specific implementation to know how debug mode is implemented. With the debugmode activated the FPGA generates the values for the PVs.

```
record(bo,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-
DEBUGMODE"){
    field(DESC,"Controls FPGA DebugMode")
    field(DTYP,"asynInt32")
    field(OUT,"@asyn(RIO_$(MODULEIDX),0)debug")
    field(PINI,"NO")
    field(ZNAM,"OFF")
    field(ONAM,"ON")
}
```

6.4.1.6 *Device Quality Status*

This record retrieves the status of the data acquisition devices in terms of quality. For instance a DAQ device can be measuring poor signals degraded by a poor S/N ratio. DevQualityStatus can give the user this information. The developer has to define the values obtained with this PV.

```
record(ai,"$(CBS1)-$(CBS2)-HWCF-
R$(RACK)C$(CHASSIS)B$(BOARD):DEVQUALITYSTATUS"){
    field(DESC,"Info of errors in acquisition process")
    field(DTYP,"asynInt32")
    field(INP,"@asyn(RIO_$(MODULEIDX),0)DevQualityStatus")
    field(SCAN,"1 second")
    field(PINI,"NO")
}
```

6.4.2 Records Used by Point by Point Profile

Point by Point profile can be used by cRIO implementations. It supports all mandatory PVs included in common resources records and adds next records instantiated in the template. This PV is included in rio-pointbypoint.template installed with irio module in */opt/codac/epics/modules/irio/db.*

6.4.2.1 *Sampling Rate*

This record controls the sampling rate used for data acquisition.

```
record(ao,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-
SR_PBP$(CHIDX)"){
     field(DESC,"Sampling rate $(CHIDX) of point by point profile")
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(CHIDX)) SamplingRate")
     field(PINI,"NO")
     field(EGU,"S/s")
     field(HOPR,"100000000")
     field(LOPR,"0")
     field(DRVH,"100000000")
     field(DRVL,"0")
     field(PREC,"6")
}
```

6.4.3 Records Used by Data Acquisition Profile

Data acquisition profile can be used by cRIO and FlexRIO implementations. It supports all mandatory PVs included in common resource records and adds next records instantiated in the next templates. PVs included in rio-dataaquisition.template are: GroupEnable, SamplingRate, DMAsOverflow and DecimationFactor. Waveform Channel PV is included in rio-wf.template.

6.4.3.1 *Group Enable*

This record enables writing data from DMA Target to DMA Host.

```
record(bo,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-
GROUPENABLE$(GRIDX)"){
     field(DESC,"Controls ChannelGroup$(GRIDX) enable/disable")
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(GRIDX))GroupEnable")
     field(ZNAM,"DISABLE")
     field(ONAM,"ENABLE")
     field(PINI,"NO")
}
```

6.4.3.2 *Sampling Rate*

This record sets the sampling rate

```
record(ao,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-SR$(GRIDX)"){
     field(DESC,"Sampling rate of channel group $(GRIDX)")
```

```
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(GRIDX))SamplingRate")
     field(PINI,"NO")
     field(EGU,"S/s")
     field(HOPR,"100000000")
     field(LOPR,"0")
     field(DRVH,"100000000")
     field(DRVL,"0")
     field(PREC,"6")
}
```

6.4.3.3 *DMAs Overflow*

This record retrieve the status of DMAs overflows in the device

```
record(bi,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-
DMAsOVERFLOW$(GRIDX)"){
     field(DESC,"Correct(0) or Overflow(1)")
     field(DTYP,"asynInt32")
     field(INP,"@asyn(RIO_$(MODULEIDX),$(GRIDX))DMAsOverflow")
     field(ZNAM,"CORRECT")
     field(ONAM,"OVERFLOW")
     field(PINI,"NO")
}
```

6.4.3.4 *Decimation Factor*

This record sets the decimation factor in the data acquisition process.

```
record(ao,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)G-$(BOARDTYPEIDX)-DF$(GRIDX)"){
     field(DESC,"Block Decimation factor of DMA$(GRIDX)")
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(GRIDX))DF")
     field(PINI,"NO")
     field(HOPR,"100000000")
     field(LOPR,"0")
     field(DRVH,"100000000")
     field(DRVL,"0")
     field(PREC,"6")
}
```

6.4.3.5 *Waveform Channel*

This record acquires data from DMA Host in arrays of $(NSAMPLE) elements every interrupt generated from the asyndriver.

```
record(waveform,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-CH$(CHIDX)"){
     field(DESC,"AI waveform $(CHIDX)")
     field(DTYP,"asynFloat32ArrayIn")
     field(INP,"@asyn(RIO_$(MODULEIDX),$(CHIDX))CH")
     field(FTVL,"FLOAT")
     field(PINI,"NO")
     field(NELM,"$(NSAMPLE)")
     field(PREC,"6")
     field(SCAN,"I/O Intr")
}
```

6.4.4 Records for optional resources

6.4.4.1 *Analog Input*

This record has the value corresponding to an analog input.

```
record (ai, "$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-
AI$(CHIDX)"){
     field(DESC,"Analog Input channel $(CHIDX) value")
     field(DTYP,"asynFloat64")
     field(INP,"@asyn(RIO_$(MODULEIDX),$(CHIDX))AI")
     field(SCAN,".1 second")
     field(PREC,"6")
}
```

6.4.4.2 *Auxiliary Analog Input*

This record has the value corresponding to an auxiliary analog input.

```
record (ai, "$(CBS1)-$(CBS2)-HWCF-R$(RACK)C$(CHASSIS)B$(BOARD):auxAI$(CHIDX)"){
     field(DESC,"Aux Analog Input of channel $( CHIDX)")
     field(DTYP,"asynInt32")
     field(INP,"@asyn(RIO_$(MODULEIDX),$(CHIDX))auxAI")
     field(SCAN,".1 second")
     field(PREC,"6")
}
```

6.4.4.3 *Analog Output*

This record value is the output of the analog output channel in the I/O Module. Only if AOEnable is set to TRUE the AO value will be outputted throw the analog output channel of the I/O Module.

```
record (ao, "$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-AO$(CHIDX)"){
     field(DESC,"Analog Output channel $(CHIDX) value")
     field(DTYP,"asynFloat64")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(CHIDX))AO")
     field(HOPR,"$(HOPR)")
     field(LOPR,"$(LOPR)")
     field(DRVH,"$(DRVH)")
     field(DRVL,"$(DRVL)")
     field(PREC,"6")
}
```

The AOEnable record enables/disables the output of the analog output channel in the I/O Module. TRUE enables the output.

```
record(bo,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-AOENABLE$(CHIDX)"){
     field(DESC,"Analog Output channel $(CHIDX) enable")
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(CHIDX))AOEnable")
     field(PINI,"NO")
     field(ZNAM,"DISABLE")
```

```
     field(ONAM,"ENABLE")
}
```

6.4.4.4 *Auxiliary Analog Output*

This record allows writing an internal register in the FPGA (auxiliary control).

```
record (ao, "$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-auxAO$(CHIDX)"){
     field(DESC,"Analog Output aux channel $(CHIDX) value")
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(CHIDX))auxAO")
     field(PREC,"6")
     field(HOPR,"$(HOPR)")
     field(LOPR,"$(LOPR)")
     field(DRVH,"$(DRVH)")
     field(DRVL,"$(DRVL)")
}
```

6.4.4.5 *Digital Input*

This record has the value of the analog input

```
record (bi, "$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-DI$(CHIDX)"){
     field(DESC,"Digital Input channel $(CHIDX) value")
     field(DTYP,"asynInt32")
     field(INP,"@asyn(RIO_$(MODULEIDX),$(CHIDX))DI")
     field(SCAN,".1 second")
     field(ZNAM,"False")
     field(ONAM,"True")
}
```

6.4.4.6 *Auxiliary Digital Input*

This record has the value of the auxiliary analog input

```
record (bi, "$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-auxDI$(CHIDX)"){
     field(DESC,"Digital Input aux channel $(CHIDX) value")
     field(DTYP,"asynInt32")
     field(INP,"@asyn(RIO_$(MODULEIDX),$(CHIDX))auxDI")
     field(SCAN,".1 second")
     field(ZNAM,"False")
     field(ONAM,"True")
}
```

6.4.4.7 *Digital Output*

This record writes a value on the digital output

```
record (bo, "$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-DO$(CHIDX)"){
     field(DESC,"Digital Output channel $(CHIDX)")
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(CHIDX))DO")
     field(ZNAM,"False")
     field(ONAM,"True")
}
```

6.4.4.8 *Auxiliary Digital Output*

This record writes a value on the auxiliary digital output

```
record (bo, "$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-auxDO$(CHIDX)"){
     field(DESC,"Digital Output aux channel $(CHIDX) value")
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(CHIDX))auxDO")
     field(ZNAM,"False")
     field(ONAM,"True")
}
```

6.4.4.9 *Signal generator*

6.4.4.9.1 Signal Generator Update Rate

This record allows to write the update rate in the signal generator.

```
record(ao,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-
SGUPDATERATE$(CHIDX)"){
     field(DESC,"Signal Generator Update Rate $(CHIDX)")
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(CHIDX))SGUpdateRate")
     field(PREC,"6")
     field(PINI,"NO")
     field(HOPR,"100000000")
     field(LOPR,"0")
     field(DRVH,"100000000")
     field(DRVL,"0")
}
```

6.4.4.9.2 Signal Generator Frequency

This record sets the frequency of the signal generated in the FPGA.

```
record(ao,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-SGFREQ$(CHIDX)"){
     field(DESC,"Signal Generator freq $(CHIDX)")
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(CHIDX))SGFreq")
     field(PREC,"6")
     field(PINI,"NO")
     field(HOPR,"100000000")
     field(LOPR,"0")
     field(DRVH,"100000000")
     field(DRVL,"0")
}
```

6.4.4.9.3 Signal Generator Phase Control

This record sets the phase of the signal generated in the FPGA.

```
record(ao,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-SGPHASE$(CHIDX)"){
     field(DESC,"Signal Generator phase control $(CHIDX)")
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(CHIDX))SGPhase")
     field(PINI,"NO")
     field(PREC,"6")
     field(HOPR,"100000000")
     field(LOPR,"0")
```

```
     field(DRVH,"100000000")
     field(DRVL,"0")
}
```

6.4.4.9.4 Signal Generator Amplitude

This record sets the amplitude of the signal generated in the FPGA.

```
record(ao,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-SGAMP$(CHIDX)"){
     field(DESC,"DSS accumulator increment $(CHIDX)")
     field(DTYP,"asynFloat64")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(CHIDX))SGAmp")
     field(PINI,"NO")
     field(PREC,"6")
     field(HOPR,"100000000")
     field(LOPR,"0")
     field(DRVH,"100000000")
     field(DRVL,"0")
}
```

6.4.4.9.5 Signal Generator Signal Type

This record configures the shape of the signal generated in FPGA. It can be DC, SINE, SQUARE or TRIANGLE.

```
record(mbbo,"$(CBS1)-$(CBS2)-HWCF:$(BOARDTYPE)-$(BOARDTYPEIDX)-
SGSIGNALTYPE$(CHIDX)"){
     field(DESC,"Type of signal generated $(CHIDX))")
     field(DTYP,"asynInt32")
     field(OUT,"@asyn(RIO_$(MODULEIDX),$(CHIDX))SGSignalType")
     field(PINI,"NO")
     field(ZRVL,"0")
     field(ONVL,"1")
     field(TWVL,"2")
     field(THVL,"3")
     field(ZRST,"DC")
     field(ONST,"SINE")
     field(TWST,"SQUARE")
     field(THST,"TRIANGLE")
}
```

7 NI-RIO EPICS DEVICE DRIVER USE IN I&C APPLICATIONS

This chapter describes how to use the NI-RIO EPICS driver (Asyn port driver) in I&C application. As with other standard boards, the SDD tools know about the RIO devices, and can be selected when adding I/O boards to a fast controller. This section describes mandatory steps that will allow the correct software initialization and the creation of EPICS variables interfaced with the device driver. The device driver is implemented using the asynDriver [RD21]. As a result, the record fields need to follow the asyn syntax, specifically for the DTYP, INP and OUTfields.

- DTYP = [asynInt32|asynInt32ArrayIn|asynInt32ArrayOut|asynFloat64]
- INP,OUT = @asyn(RIO_$(MODULEIDX), $(CHIDX)) $(ASYNCOMMAND)

The SDD tools assist the user with the configuration of these fields. The procedure, illustrated here with SDD Editor [RD23], details usage of NI-RIO EPICS Device Driver in the application.

1. Create/edit an I&C Project.
2. Create/edit the fast controller that will control the RIO device.
3. Add a RIO device to this controller.
    a. At boot-up time, the Linux driver automatically creates one resource name for a device presented in the system. The names follow the syntax RIO<x>. Here x is the board index of RIO device, starting from 0.
4. Instantiate Templates
5. Add board functional variables individually without the Template as follows:
    a. Create/edit the variable.
    b. Choose the record type according to the functions described in the section 6.2.
    c. Set the deployment unit to the fast controller.
    d. Edit the predefined attributes
        i. The index of the board [MODULEIDX]
        ii. The command [ASYNCOMMAND]
        iii. The index of the channel [CHIDX]

7.1 Makefile Configuration

With usage of the NI-RIO EPICS driver in the application, the following code will be added to <top>/<application_name>App/src/Makefile by SDD Tool kit:

```
# <application_name>.dbd will be made up from these files:
<application_name>_DBD += base.dbd
<application_name>_DBD += asyn.dbd
<application_name>_DBD += irioAsyn.dbd
# Add all the support libraries needed by this IOC
<application_name>_LIBS += asyn
<application_name>_LIBS += irio-epics
```

7.2 Development of the Sample Application in CCSv5.1 or higher for cRIO

This example describes the basic idea of I&C application development with the RIO device using SDD Editor (CCS5.1 or higher [RD24]). The procedure below describes how to create a project for SDD using a compactRIO system. The objective is to acquire analog signals, generate analog output signals and interface with digital I/O elements. This example is based on a LabVIEW for FPGA implementation providing access to the following resources. The resources implemented in the FPGA are identified using labels. In this example the label used are displayed in column "identification". (see Table 30).

Table 30: Summary of channels used in compactRIO system

id	Device	Resources used	Identification	Comments
Chassis	NI9159	auxDO0,auxDO1 auxDIO, auxDI1 auxAO0,auxAO1 auxAI0,auxAI1	auxDO0,auxDO1 auxDIO, auxDI1 auxAO0,auxAO1 auxAI0,auxAI1	Aux resources are terminal (control and indicators, registers) implemented in the FPGA. These are not connected to physical signals
AO	NI9264	AO0, AO1,AO2	AO0,AO1,AO2	Three analog outputs generated using the compactRIO module NI9264.
AI	NI9205	AI0,AI1,AI2	AI0,AI1,AI2	Three analog inputs measured using the compactRIO module NI9205.
DIO	NI9401	DIO0 (as output) DIO4 (as input)	DO0 DI0	One digital output and one digital input in NI9401
DO	NI9477	DO0	DO1	One digital output
DI	NI9426	DIO	DI1	One digital input
DO	NI9476	DO0	DO2	One digital output
DI	NI9425	DI0	DI2	One digital input

The steps are:

- Open SDD Editor as depicted in Fig. 55 or execute the following command in a terminal window

```
& sdd-editor &
```

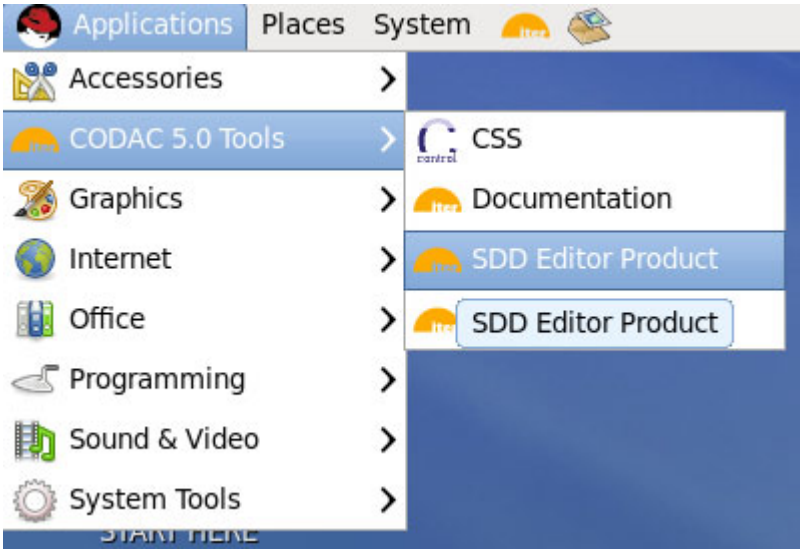

Fig. 55 Opening the SDD Editor Product

- Selection of the workspace (see Fig. 56)

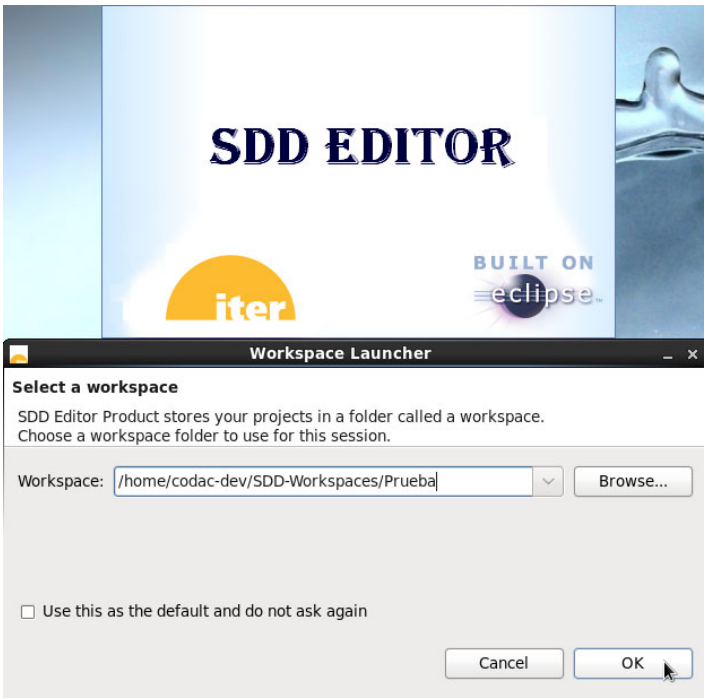

Fig. 56 Selecting the workspace

- Creation of a Project in SDD Editor. The process to accomplish it can be chekcked in the SDD Editor User Manual [RD23], see Fig. 57.

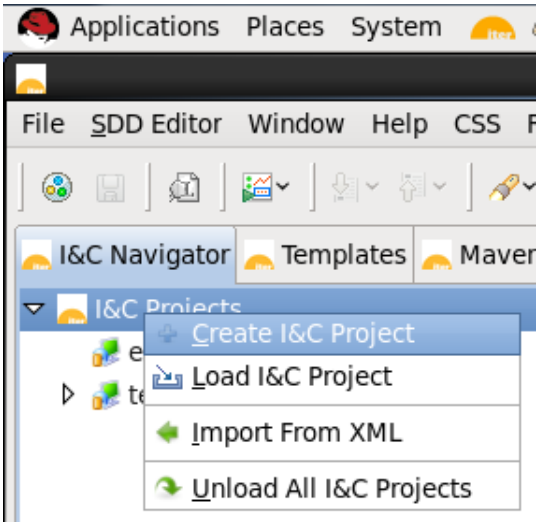

Fig. 57 Creating a new project.

- Name the I&C Project *epics-cRIO-sample*.
- Set the following parameters:
  - Select PBS = 45-CODAC
  - Select CBS1 = TEST (from drop down list).
  - CBS2=cRIO and CBS3=FUN (see Fig. 58)

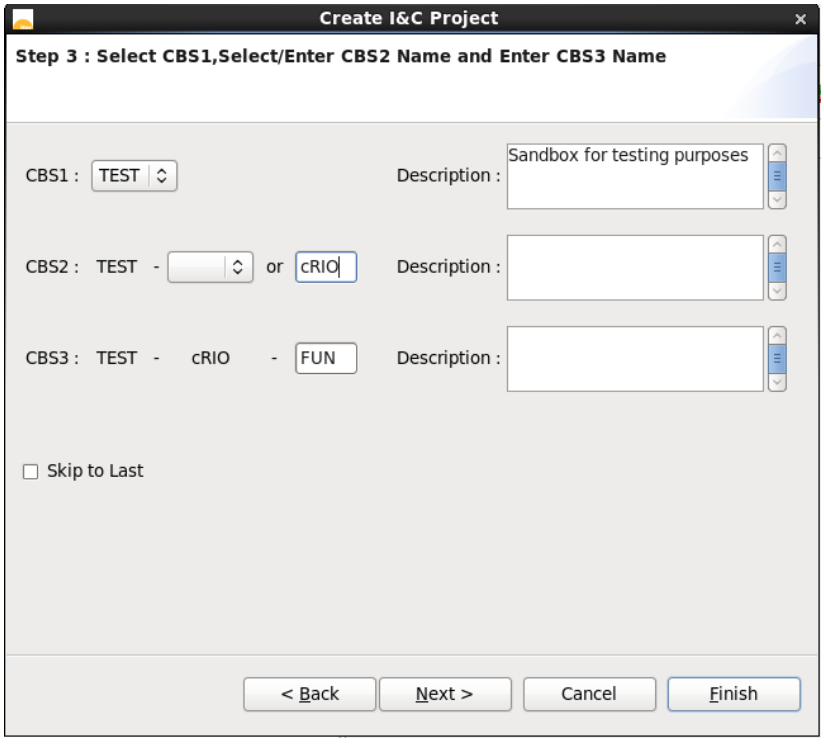

Fig. 58 Selecting the CBS1, CBS2 and FUN.

  - PSH Name = 45TEST-PSH-001

- PCF Name = 45TEST-PCF-001
- Add I/O Module to the Fast controller
  - Name=45TEST-IOM-001
  - Index=0
  - I/O Module Type-compactRIO ->compactRIO

- Finally the configuration is depicted in Fig. 59.

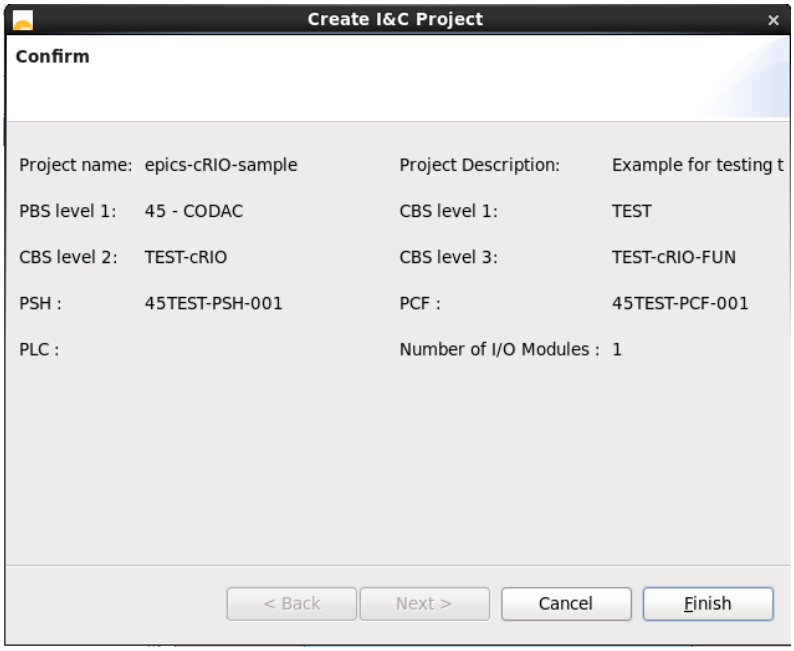

Fig. 59 Configuration summary in SDD Editor

- Navigate to in the I/O Module under the corresponding Fast Controller (see Fig. 60) and double click on it.

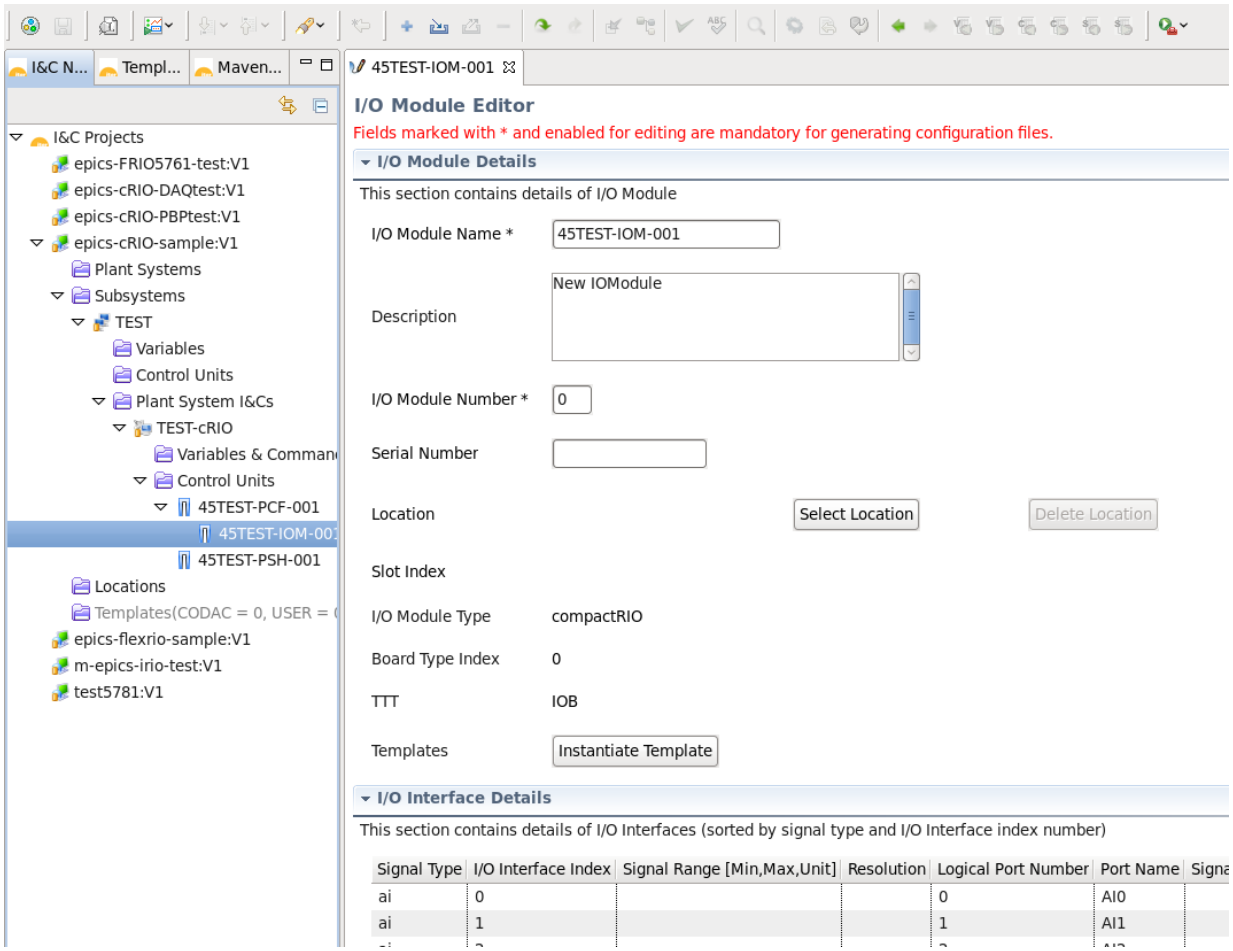

Fig. 60 Selecting the module 45TEST-IOM-001

- The instantiation of the module Template *compactRIO_module* with default parameters (see Fig. 61) is done clicking in *Instantiate Template*. This template provides de basic PVs to interact with the RIO device. These PVs are common in all RIO-based applications.

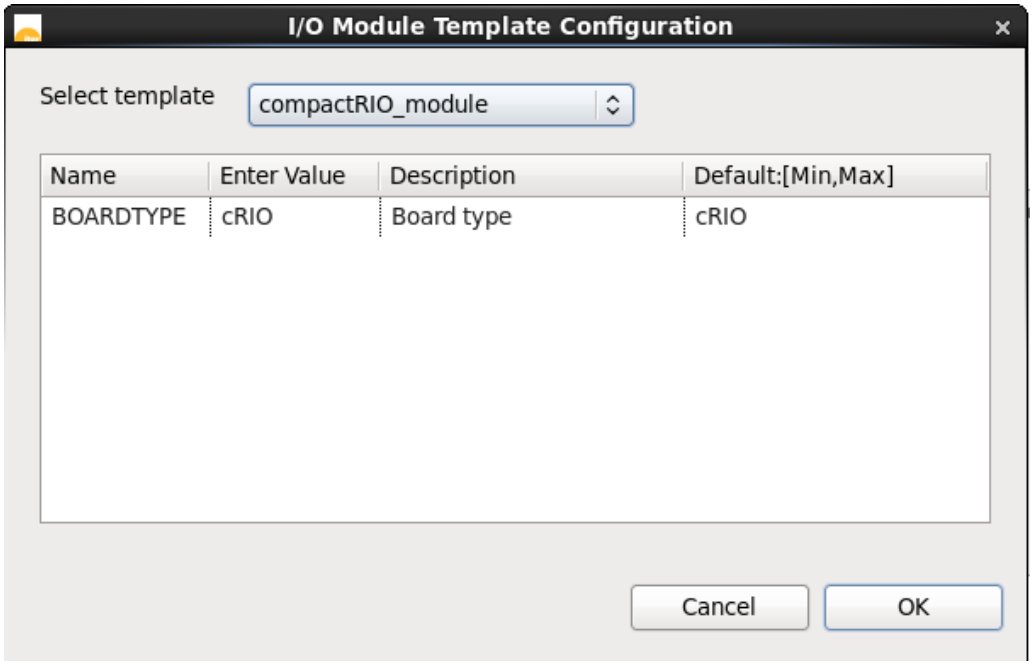

Fig. 61 Instantiation of compactRIO_module template

These PVs are depicted in Fig. 62:

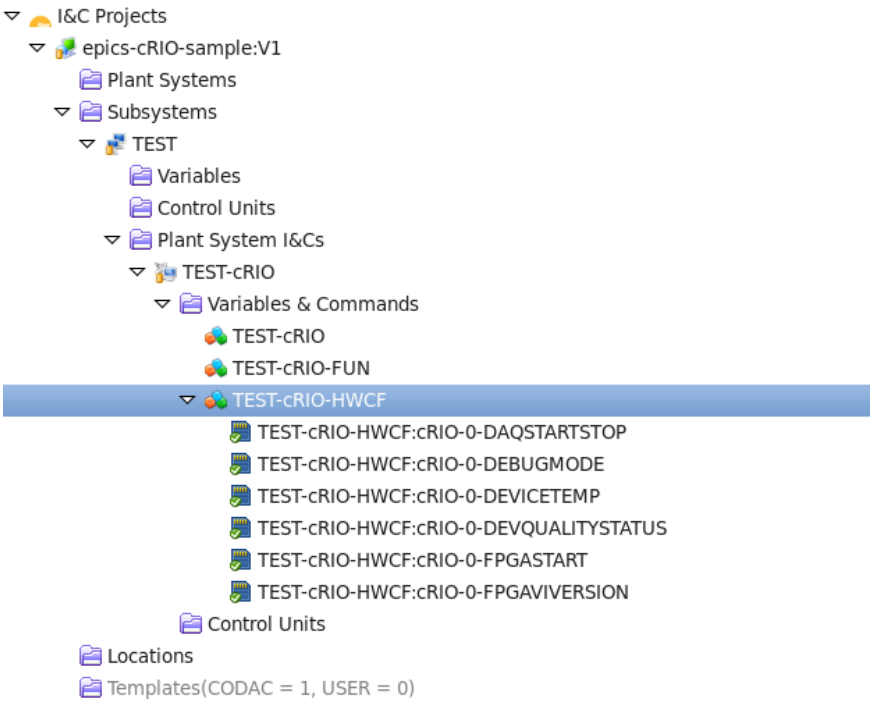

Fig. 62 HWCF PVs generated when instantiating cRIO module template

The compactRIO_PointByPoint template has to be instantiated in this point. This template adds the sampling rate PV for the application. A new PV is added to TEST-cRIO-HWCF in Variables&Commands. This is the template for the first group of channels (and the only one implemented in the FPGA), then CHIDX =0.

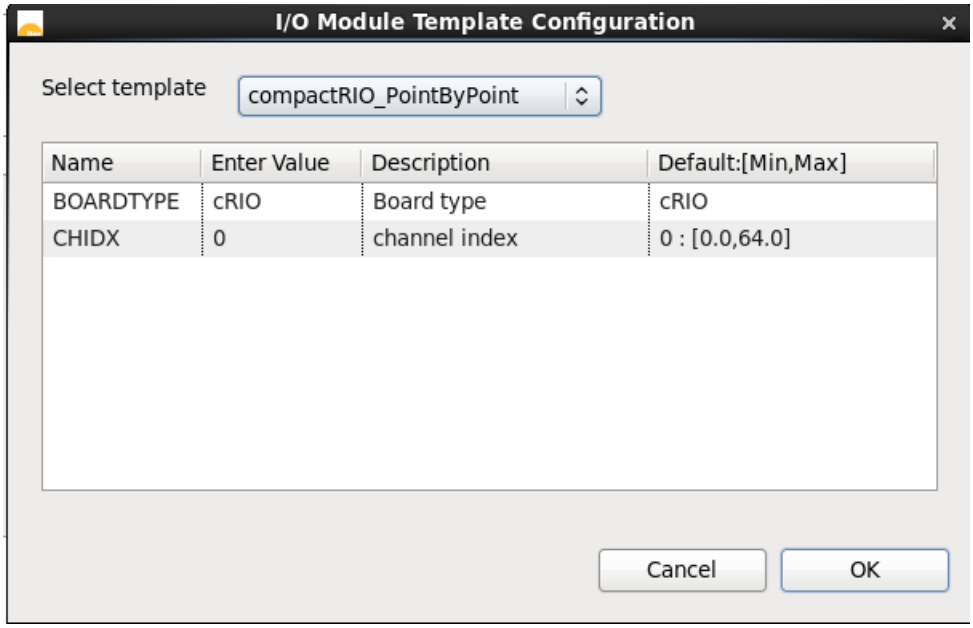

Fig. 63 compactRIO_PointByPoint template

The next template to be instantiated is compactRIO_AI template, see Fig. 64. As this example is going to use 3 channels the template will be instantiated three times.

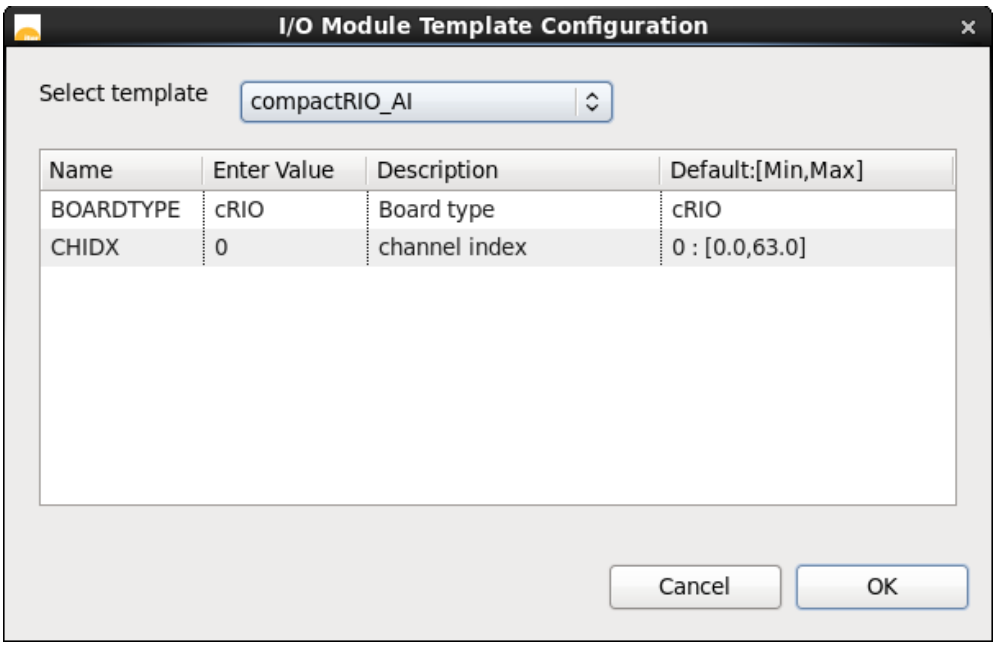

Fig. 64 Channel instantiation

This instantiation process is repeated for the different templates of analog output, digital input, output, etc (see Fig. 65 to Fig. 71) as summarized in Table 31

Table 31: Summary of templates used in the example

Template	Number of times to be instantiated	Comments
compactRIO_AO	3	There are 3 analog outputs
compactRIO_DI	3	3 digital inputs
compactRIO_DO	3	3 digital outputs
compactRIO_auxAI	2	2 auxiliaries analog inputs
compactRIO_auxAO	2	2 auxiliaries analog outputs
compactRIO_auxDI	2	2 auxiliary digital inputs
compactRIO_auxDO	2	2 auxiliary digital outputs

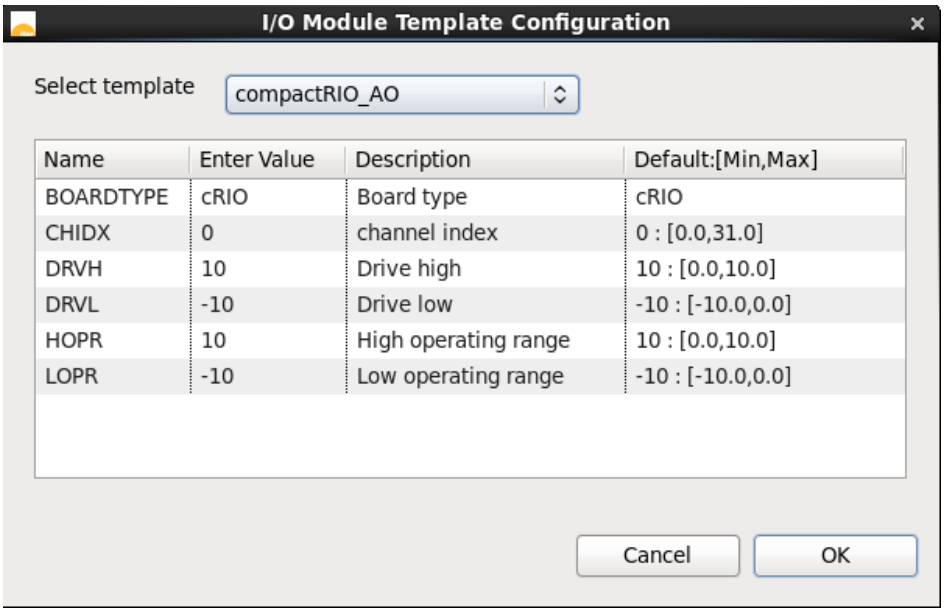

Fig. 65 Analog output template

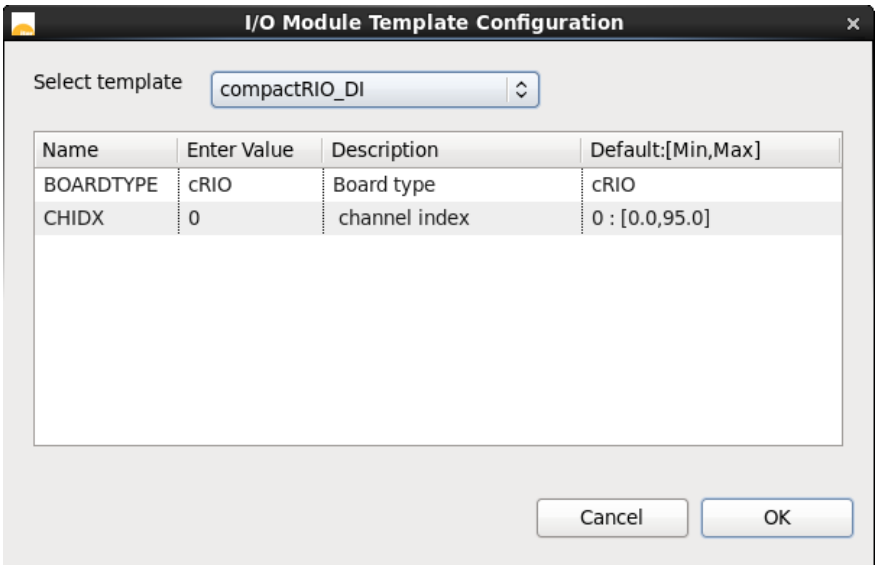

Fig. 66 Digital Input Template

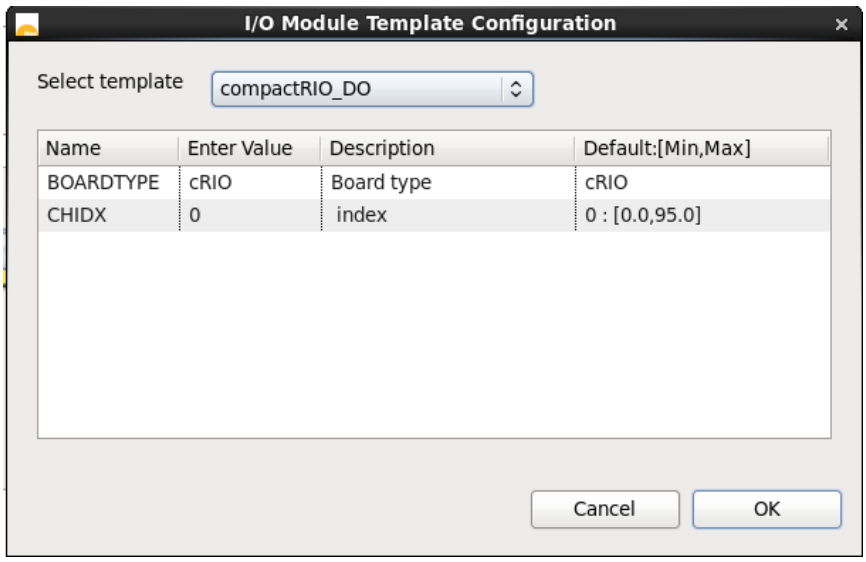

Fig. 67 Digital output template

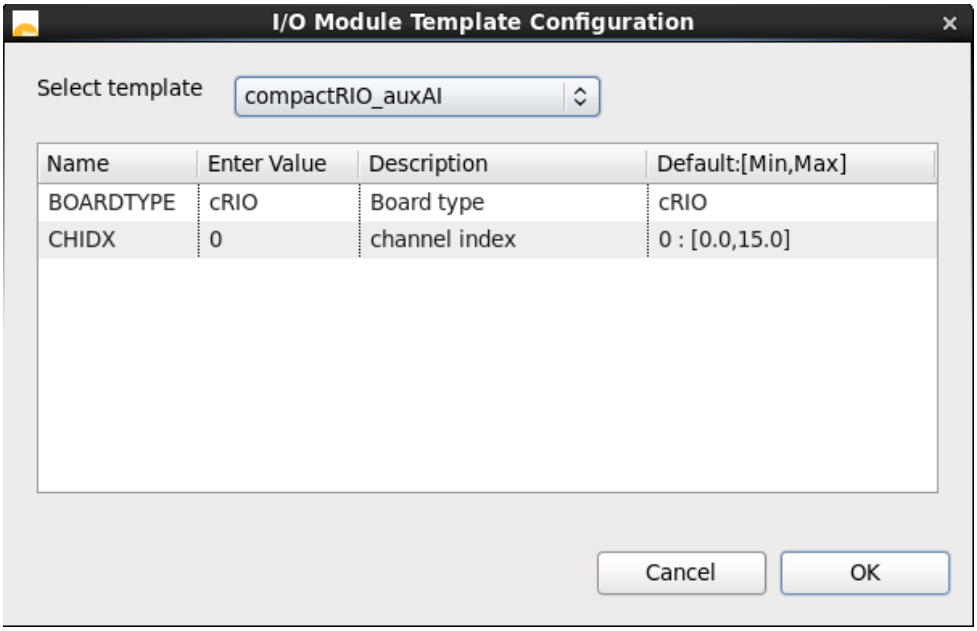

Fig. 68 Auxiliary analog input template

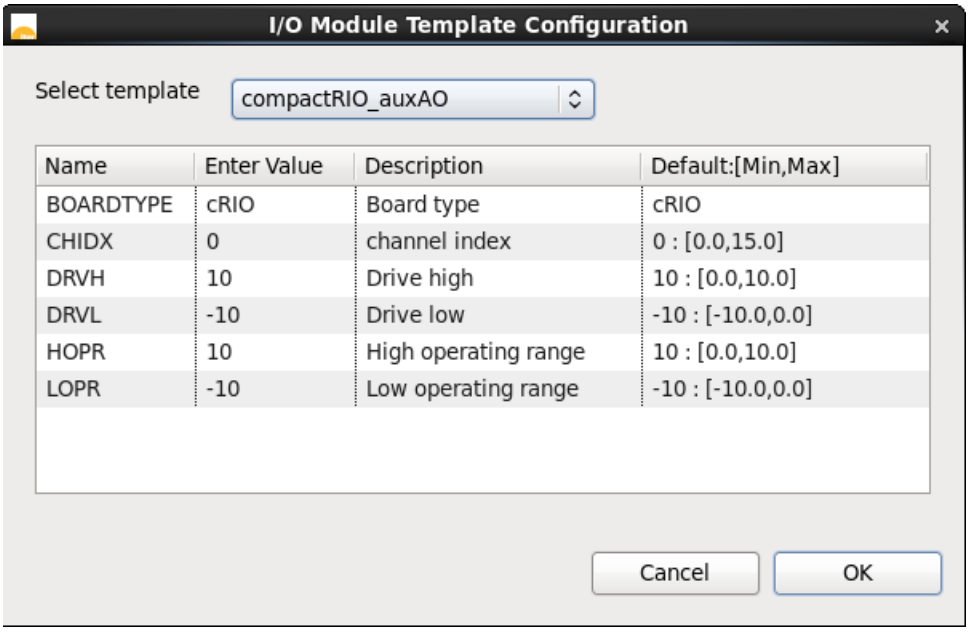

Fig. 69 Auxiliary analog output template

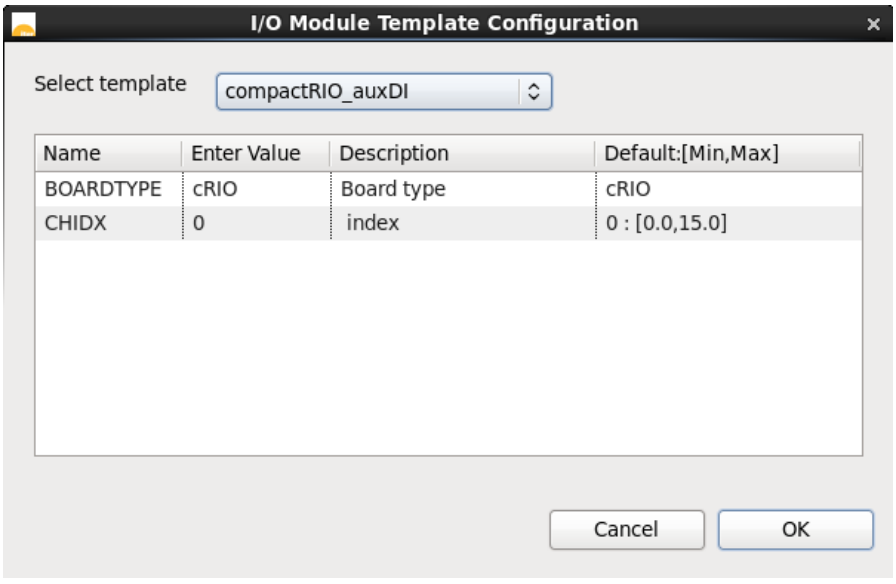

Fig. 70 Auxiliary digital input

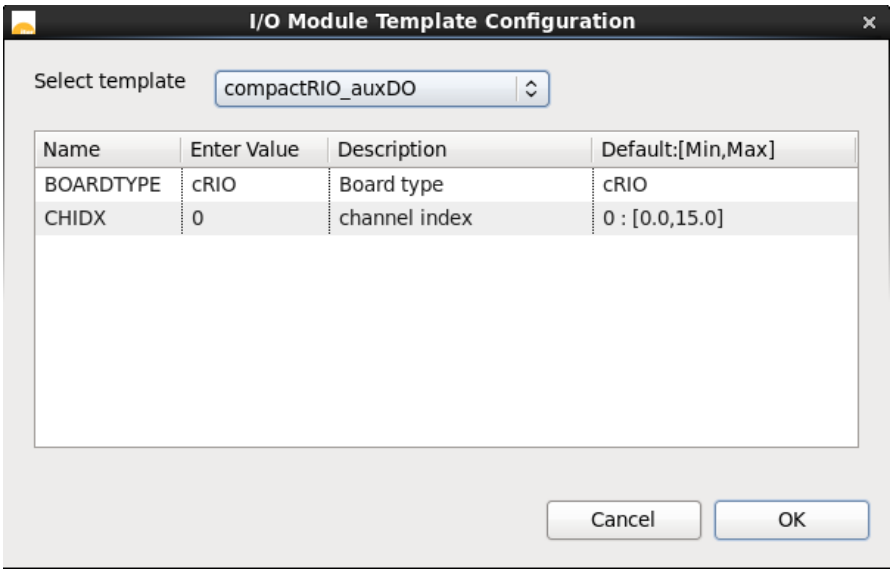

Fig. 71 Auxiliary digital output

Once the template instantiation has been completed, the next step is the completeness validation of the design (seeFig. 72). The output of this command should be that all the unit is correct although some warnings can appear related with the definition of Ethernet interfaces.

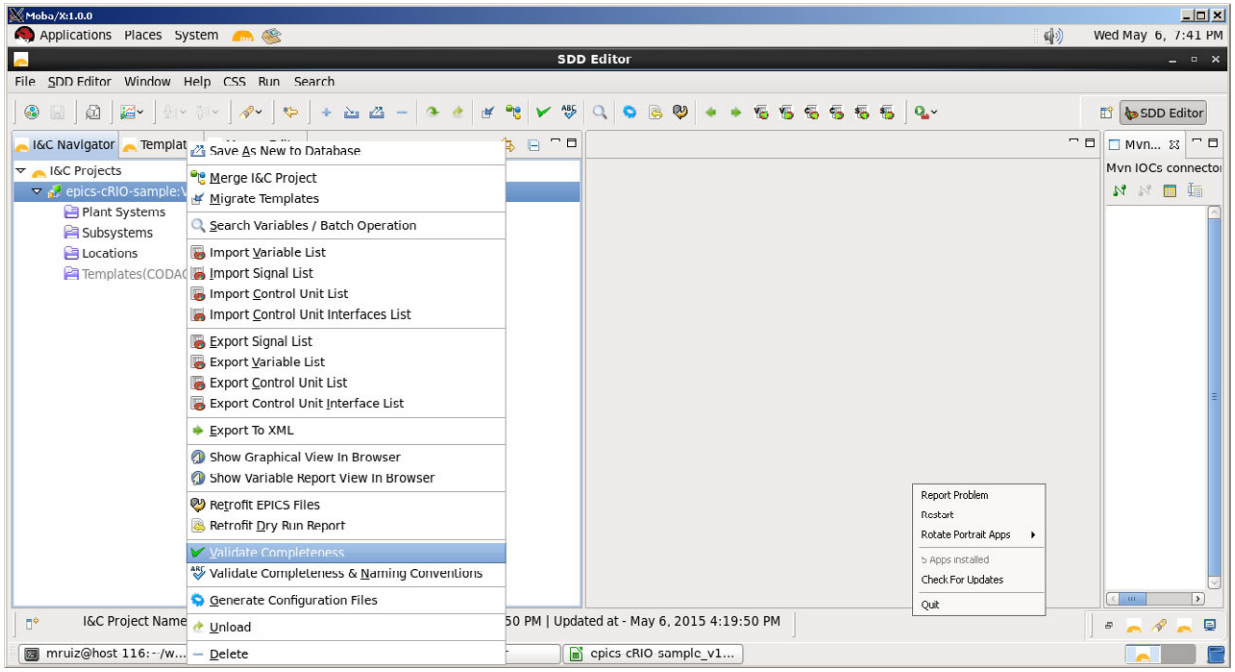

Fig. 72 Validating the project

Now the configuration files can be generated, this files consist of the required files to launch the EPICS application to control the instantiated hardware. Fig. 73 depicts the path location of the output files.

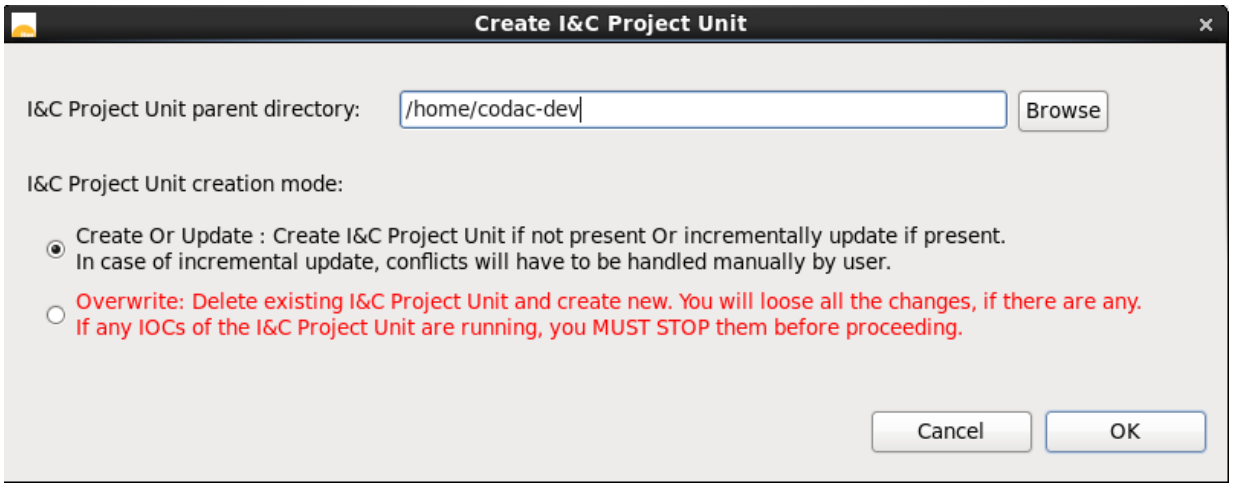

Fig. 73 Folder for unit creation

Once the unit is generated, the userPreDriverConf.cmd file has to be edited to include the EPICS Channel Access depth and the aynDriver function described in Chapter 6 to initialize the RIO device. The lines to add in the userPreDriverConf.cmd file are:

```
epicsEnvSet("EPICS_CA_MAX_ARRAY_BYTES","170000")
nirioinit("RIO_0","019ED079","NI 9159","crioPBPDAQ","V1.0",1)
```

This provides the initialization of RIO device with serial number 0x019ED079 corresponding to a NI9159 compactRIO device with the configuration defined.

Using SDD Editor tools, the unit can be compiled by right click and selecting Build->Compile.

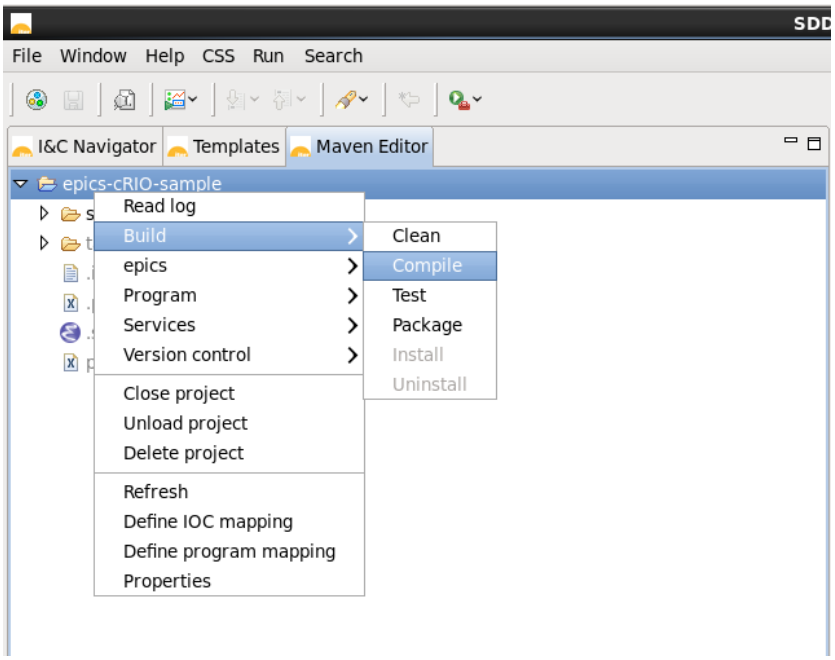

Fig. 74 Compiling the unit with Maven Editor

The test TEST-cRIO-PCF0CORE IOC now is ready to be run selecting epics->Run IOCs (see Fig. 75 and Fig. 76). Select properly the name of machine running the IOC.

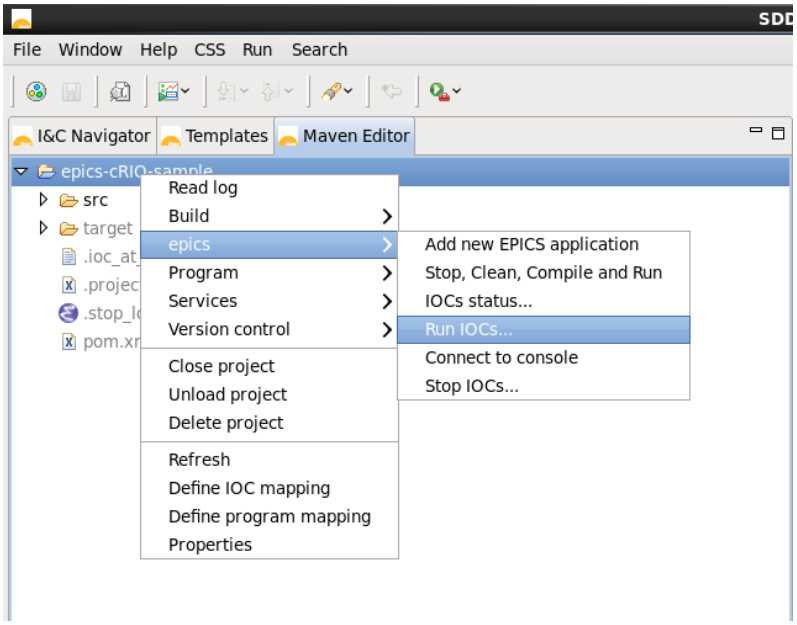

Fig. 75 Running the fast controller

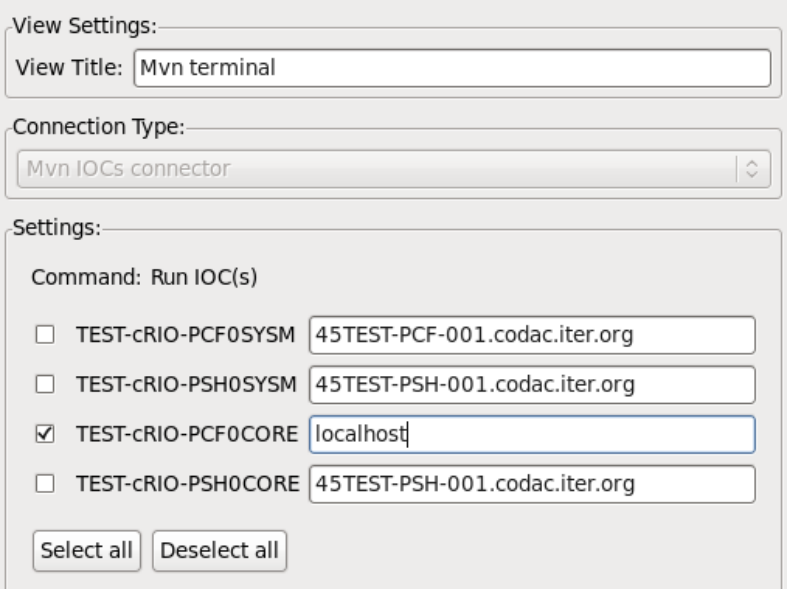

Fig. 76 Running the PCF0CORE IOC

Now, the IOC is running in the fast controller. The status of the IOC can be checked In order to verify if is running or not. The OPI panel SDD TEST-SDDMain.opi of the application in the folder main\boy\sdd\ is automatically created with a basic HMI to the system.

The sequences of actions to verify that IOC is running correctly are:

1. cRIO-0-FPGASTART to ON
2. cRIO-O-AOEnable0,1,2 to ENABLE
3. cRIO-0-DAQSTARTSTOP to ON
4. Set PVS cRIO-0-AO0 to 1.0, cRIO-0-AO1 to -1.0 and cRIO-0-AO2 to 1.5.
5. The PVS cRIO-0-AI0, cRIO-0-AI1, cRIO-0-AI2 should display values close to 1.0, -1.0 and 1.5.

Fig. 77 and Fig. 78 depict the automatically created HMI OPI Panel to handle the DAQ device.

These tests demonstrate the full integration of the CompactRIO platform in CODAC Core System using EPICS, AsynDriver and SDD tool in order to automate the creation of applications for this technology.

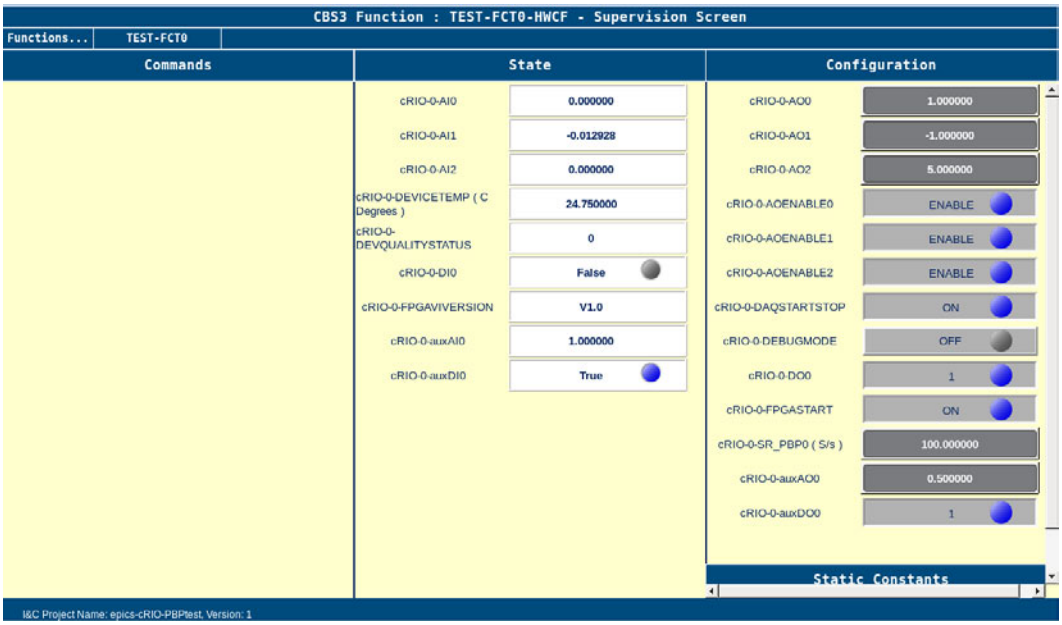

Fig. 77 PCF0CORE IOC OPI Panel

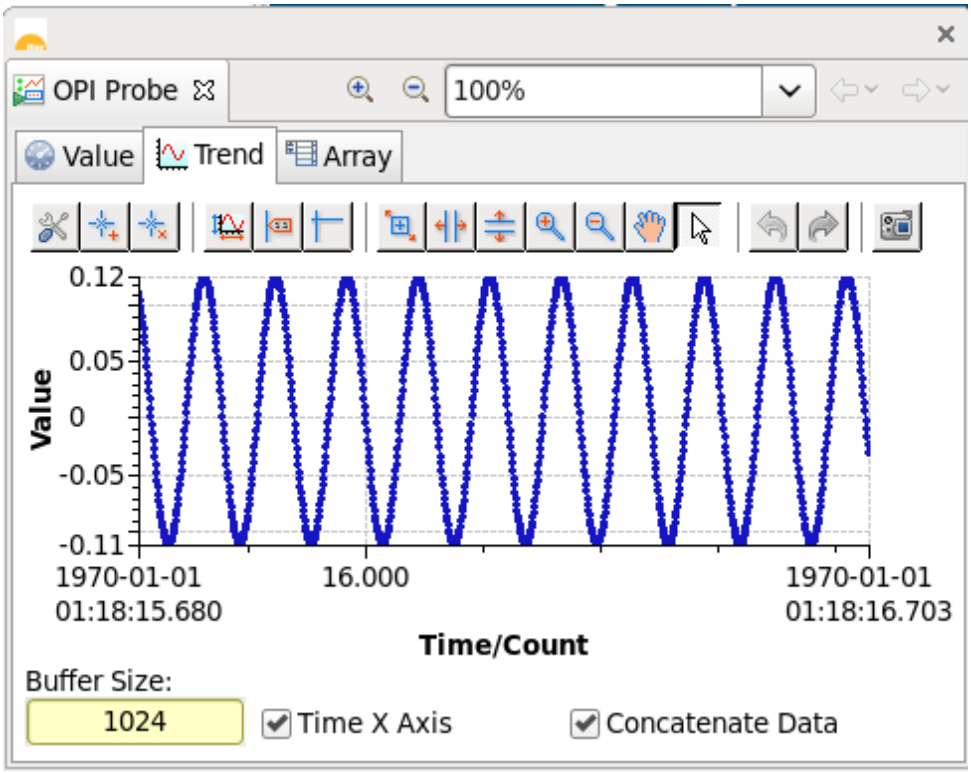

Fig. 78 PCF0CORE IOC OPI Panel Monitoring Analog Input Acquisition Channel

8 RESULTS AND CONCLUSIONS

The intention of this document lies on the integration of National Instruments CompactRIO devices to be used as part of the ITER Hardware Catalogue through the Control Data Access and Communication Core System created for the Instrumentation and Control of the ITER experiment. The different sections of this document cover the steps accomplished to successfully achieve such implementation. The subsequent list enumerates the achieved tasks developed in this project.

- ✓ Preliminary study of the CompactRIO platform functionalities, characteristics and the mechanisms offered by LabVIEW FPGA for its configuration to define a basic architecture model that enables the specific input/output resources management.
- ✓ Establish an ensemble of name conventions and rules to standardize the data acquisition systems hardware development to easily being integrated in EPICS and CCS.
- ✓ Creation of LabVIEW FPGA DAQ templates exemplifying such naming and rules conventions to include as part of the IRIO project in order to guide the ITER hardware developers.
- ✓ Definition of a basic model to represent the CompactRIO system in EPICS describing its records and process variables to be used.
- ✓ Implementation of the asynDriver device support to link the defined process variables to the hardware implemented in the FPGA.
- ✓ Creation of stand-alone C language applications using the aforementioned hardware templates and creation of an EPICS IOC that uses the implemented driver in order to test the EPICS and asynDriver implementation.
- ✓ Implementation of the configuration files for SDD-editor tool in order to automatize the creation and management of all CompactRIO DAQ resources implemented in the FPGA.

9 REFERENCES

[RD1] ITER Catalog of I&C products - Fast Controllers (345X28 v2.4)

[RD2] PXI Systems Alliance

[RD3] CODAC Core System User Manual. (43PSH9 v3.4)

[RD4] Instrumentation and Applied Acoustics Research Group

[RD5] IRIO Library User's Manual

[RD6] EPICS Application Developer's Guide (http://www.aps.anl.gov/epics/base/R3-14/11-docs/AppDevGuide.pdf)

[RD7] LabVIEW TM FPGA Course Manual. Course Software Version 2009. August 2009 Edition. Part Number 372510C-01.

[RD8] NI LabVIEW for CompactRIO. http://www.ni.com/pdf/products/us/fullcriodevguide.pdf

[RD9] NI R Series Multifunction RIO User Manual. Edition Date: June 2009.Part Number: 370489G-01. http://www.ni.com/pdf/manuals/370489g.pdf

[RD10] The NI LabVIEW High-Performance FPGA Developer's Guide. http://www.ni.com/tutorial/14600/en/

[RD11] http://zone.ni.com/reference/en-XX/help/370984T-01/criodevicehelp/module_ids/

[RD12] NI9159 Operating Instructions and Specifications.

[RD13] NI9205 Operating Instructions and Specifications.

[RD14] NI9264 Operating Instructions and Specifications.

[RD15] NI9401 Operating Instructions and Specifications.

[RD16] NI9477 Operating Instructions and Specifications.

[RD17] NI9426 Operating Instructions and Specifications.

[RD18] NI9425 Operating Instructions and Specifications.

[RD19] NI9476 Operating Instructions and Specifications.

[RD20] IRIO Design Rules for LabVIEW for FPGA (QQMYTY v1.2)

[RD21] Asyndriver http://www.aps.anl.gov/epics/modules/soft/asyn/

[RD22] NI-RIO EPICS Device Driver User Manual (RAJ9P8 v1.0)

[RD23] CODAC Core System Self-Description Data Editor User Manual (32Z4W2 v.8.0)

[RD24] CODAC Core System Application Development Manual (33T8LW v5.0)

[RD25] Integration of a data acquisition system based on FlexRIO technology with EPICS

[RD26] ITER CODAC Acronyms List (2LT73V v2.0)

[RD27] NI-RIO Device Driver User Manual (LW3UFH v2.4)

Appendix A. C API GENERATED FILES FOR CRIO TEMPLATES

a. CRIO POINT BY POINT HEARDERFILE

```
/*
 * Generated with the FPGA Interface C API Generator 13.0.0
 * for NI-RIO 13.0.0 or later.
 */

#ifndef __NiFpga_cRIOIO_9159_h__
#define __NiFpga_cRIOIO_9159_h__

#ifndef NiFpga_Version
   #define NiFpga_Version 1300
#endif

#include "NiFpga.h"

/**
 * The filename of the FPGA bitfile.
 *
 * This is a #define to allow for string literal concatenation. For
example:
 *
 *    static const char* const Bitfile = "C:\\"
NiFpga_cRIOIO_9159_Bitfile;
 */
#define NiFpga_cRIOIO_9159_Bitfile "NiFpga_cRIOIO_9159.lvbitx"

/**
 * The signature of the FPGA bitfile.
 */
static const char* const NiFpga_cRIOIO_9159_Signature =
"0AAC99310A046DA5E746222A2B4D08BF";

typedef enum
{
   NiFpga_cRIOIO_9159_IndicatorBool_DI0 = 0x813A,
   NiFpga_cRIOIO_9159_IndicatorBool_DI1 = 0x813E,
   NiFpga_cRIOIO_9159_IndicatorBool_DI2 = 0x816E,
   NiFpga_cRIOIO_9159_IndicatorBool_InitDone = 0x8176,
   NiFpga_cRIOIO_9159_IndicatorBool_auxDI0 = 0x814A,
   NiFpga_cRIOIO_9159_IndicatorBool_auxDI1 = 0x814E,
   NiFpga_cRIOIO_9159_IndicatorBool_cRIOModulesOK = 0x8182,
} NiFpga_cRIOIO_9159_IndicatorBool;

typedef enum
{
   NiFpga_cRIOIO_9159_IndicatorU8_DevProfile = 0x810E,
   NiFpga_cRIOIO_9159_IndicatorU8_DevQualityStatus = 0x818A,
   NiFpga_cRIOIO_9159_IndicatorU8_Platform = 0x819A,
```

```
   NiFpga_cRIOIO_9159_IndicatorU8_SGNo = 0x8152,
} NiFpga_cRIOIO_9159_IndicatorU8;

typedef enum
{
   NiFpga_cRIOIO_9159_IndicatorI16_DevTemp = 0x817E,
} NiFpga_cRIOIO_9159_IndicatorI16;

typedef enum
{
   NiFpga_cRIOIO_9159_IndicatorU16_state = 0x81A2,
} NiFpga_cRIOIO_9159_IndicatorU16;

typedef enum
{
   NiFpga_cRIOIO_9159_IndicatorI32_AI0 = 0x8110,
   NiFpga_cRIOIO_9159_IndicatorI32_AI1 = 0x8114,
   NiFpga_cRIOIO_9159_IndicatorI32_AI2 = 0x8158,
   NiFpga_cRIOIO_9159_IndicatorI32_auxAI0 = 0x8118,
   NiFpga_cRIOIO_9159_IndicatorI32_auxAI1 = 0x811C,
} NiFpga_cRIOIO_9159_IndicatorI32;

typedef enum
{
   NiFpga_cRIOIO_9159_IndicatorU32_Fref = 0x8178,
} NiFpga_cRIOIO_9159_IndicatorU32;

typedef enum
{
   NiFpga_cRIOIO_9159_ControlBool_AOEnable0 = 0x815E,
   NiFpga_cRIOIO_9159_ControlBool_AOEnable1 = 0x8162,
   NiFpga_cRIOIO_9159_ControlBool_AOEnable2 = 0x8166,
   NiFpga_cRIOIO_9159_ControlBool_DAQStartStop = 0x8192,
   NiFpga_cRIOIO_9159_ControlBool_DO0 = 0x8132,
   NiFpga_cRIOIO_9159_ControlBool_DO1 = 0x8136,
   NiFpga_cRIOIO_9159_ControlBool_DO2 = 0x816A,
   NiFpga_cRIOIO_9159_ControlBool_DebugMode = 0x818E,
   NiFpga_cRIOIO_9159_ControlBool_auxDO0 = 0x8142,
   NiFpga_cRIOIO_9159_ControlBool_auxDO1 = 0x8146,
} NiFpga_cRIOIO_9159_ControlBool;

typedef enum
{
   NiFpga_cRIOIO_9159_ControlU16_SamplingRate0 = 0x8172,
} NiFpga_cRIOIO_9159_ControlU16;

typedef enum
{
   NiFpga_cRIOIO_9159_ControlI32_AO0 = 0x8120,
   NiFpga_cRIOIO_9159_ControlI32_AO1 = 0x8124,
   NiFpga_cRIOIO_9159_ControlI32_AO2 = 0x8154,
   NiFpga_cRIOIO_9159_ControlI32_auxAO0 = 0x8128,
   NiFpga_cRIOIO_9159_ControlI32_auxAO1 = 0x812C,
} NiFpga_cRIOIO_9159_ControlI32;

typedef enum
{
```

```
   NiFpga_cRIOIO_9159_ControlU32_LoopuSec = 0x81A4,
   NiFpga_cRIOIO_9159_ControlU32_TabControl = 0x819C,
} NiFpga_cRIOIO_9159_ControlU32;

typedef enum
{
   NiFpga_cRIOIO_9159_IndicatorArrayU8_FPGAVIversion = 0x8186,
} NiFpga_cRIOIO_9159_IndicatorArrayU8;

typedef enum
{
   NiFpga_cRIOIO_9159_IndicatorArrayU8Size_FPGAVIversion = 2,
} NiFpga_cRIOIO_9159_IndicatorArrayU8Size;

typedef enum
{
   NiFpga_cRIOIO_9159_IndicatorArrayU16_InsertedIOModulesID = 0x8194,
} NiFpga_cRIOIO_9159_IndicatorArrayU16;

typedef enum
{
   NiFpga_cRIOIO_9159_IndicatorArrayU16Size_InsertedIOModulesID = 16,
} NiFpga_cRIOIO_9159_IndicatorArrayU16Size;

#endif
```

b. cRIO Analog Signal DMA-Based DAQ Profile Headerfile

```
/*
 * Generated with the FPGA Interface C API Generator 13.0.0
 * for NI-RIO 13.0.0 or later.
 */

#ifndef __NiFpga_cRIODAQDMA_9159_h__
#define __NiFpga_cRIODAQDMA_9159_h__

#ifndef NiFpga_Version
   #define NiFpga_Version 1300
#endif

#include "NiFpga.h"

/**
 * The filename of the FPGA bitfile.
 *
 * This is a #define to allow for string literal concatenation. For
example:
 *
 *    static const char* const Bitfile = "C:\\"
NiFpga_cRIODAQDMA_9159_Bitfile;
 */
#define NiFpga_cRIODAQDMA_9159_Bitfile "NiFpga_cRIODAQDMA_9159.lvbitx"
```

```
/**
 * The signature of the FPGA bitfile.
 */
static const char* const NiFpga_cRIODAQDMA_9159_Signature =
"FF8D8A82A0973BF942C94A35D37965C4";

typedef enum
{
   NiFpga_cRIODAQDMA_9159_IndicatorBool_DI0 = 0x8156,
   NiFpga_cRIODAQDMA_9159_IndicatorBool_DI1 = 0x815A,
   NiFpga_cRIODAQDMA_9159_IndicatorBool_DI2 = 0x8172,
   NiFpga_cRIODAQDMA_9159_IndicatorBool_InitDone = 0x8192,
   NiFpga_cRIODAQDMA_9159_IndicatorBool_auxDI0 = 0x8166,
   NiFpga_cRIODAQDMA_9159_IndicatorBool_cRIOModulesOK = 0x81B6,
} NiFpga_cRIODAQDMA_9159_IndicatorBool;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_IndicatorU8_DevProfile = 0x819A,
   NiFpga_cRIODAQDMA_9159_IndicatorU8_DevQualityStatus = 0x81A6,
   NiFpga_cRIODAQDMA_9159_IndicatorU8_Platform = 0x81BA,
   NiFpga_cRIODAQDMA_9159_IndicatorU8_SGNo = 0x810E,
} NiFpga_cRIODAQDMA_9159_IndicatorU8;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_IndicatorI16_DevTemp = 0x819E,
} NiFpga_cRIODAQDMA_9159_IndicatorI16;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_IndicatorU16_DMATtoHOSTOverflows = 0x818A,
   NiFpga_cRIODAQDMA_9159_IndicatorU16_state = 0x81C2,
} NiFpga_cRIODAQDMA_9159_IndicatorU16;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_IndicatorI32_AI0 = 0x8138,
   NiFpga_cRIODAQDMA_9159_IndicatorI32_AI1 = 0x8134,
   NiFpga_cRIODAQDMA_9159_IndicatorI32_AI2 = 0x814C,
   NiFpga_cRIODAQDMA_9159_IndicatorI32_auxAI0 = 0x813C,
} NiFpga_cRIODAQDMA_9159_IndicatorI32;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_IndicatorU32_Fref = 0x8194,
   NiFpga_cRIODAQDMA_9159_IndicatorU32_SGFref0 = 0x8150,
} NiFpga_cRIODAQDMA_9159_IndicatorU32;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_ControlBool_AOEnable0 = 0x8116,
   NiFpga_cRIODAQDMA_9159_ControlBool_AOEnable1 = 0x8132,
   NiFpga_cRIODAQDMA_9159_ControlBool_AOEnable2 = 0x814A,
   NiFpga_cRIODAQDMA_9159_ControlBool_DAQStartStop = 0x81AE,
   NiFpga_cRIODAQDMA_9159_ControlBool_DMATtoHOSTEnable0 = 0x8186,
   NiFpga_cRIODAQDMA_9159_ControlBool_DO0 = 0x815E,
```

```
   NiFpga_cRIODAQDMA_9159_ControlBool_DO1 = 0x8162,
   NiFpga_cRIODAQDMA_9159_ControlBool_DO2 = 0x816E,
   NiFpga_cRIODAQDMA_9159_ControlBool_DebugMode = 0x81AA,
   NiFpga_cRIODAQDMA_9159_ControlBool_auxDO0 = 0x816A,
} NiFpga_cRIODAQDMA_9159_ControlBool;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_ControlU8_SGSignalType0 = 0x811A,
} NiFpga_cRIODAQDMA_9159_ControlU8;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_ControlU16_DMATtoHOSTSamplingRate0 = 0x8182,
   NiFpga_cRIODAQDMA_9159_ControlU16_SGAmp0 = 0x812A,
} NiFpga_cRIODAQDMA_9159_ControlU16;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_ControlI32_AO0 = 0x811C,
   NiFpga_cRIODAQDMA_9159_ControlI32_AO1 = 0x8110,
   NiFpga_cRIODAQDMA_9159_ControlI32_AO2 = 0x8144,
   NiFpga_cRIODAQDMA_9159_ControlI32_auxAO0 = 0x8140,
} NiFpga_cRIODAQDMA_9159_ControlI32;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_ControlU32_LoopuSec = 0x81C4,
   NiFpga_cRIODAQDMA_9159_ControlU32_SGFreq0 = 0x8124,
   NiFpga_cRIODAQDMA_9159_ControlU32_SGPhase0 = 0x8120,
   NiFpga_cRIODAQDMA_9159_ControlU32_SGUpdateRate0 = 0x812C,
   NiFpga_cRIODAQDMA_9159_ControlU32_TabControl = 0x81BC,
} NiFpga_cRIODAQDMA_9159_ControlU32;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_IndicatorArrayU8_DMATtoHOSTFrameType = 0x817A,
   NiFpga_cRIODAQDMA_9159_IndicatorArrayU8_DMATtoHOSTSampleSize =
0x817E,
   NiFpga_cRIODAQDMA_9159_IndicatorArrayU8_FPGAVIversion = 0x81A2,
} NiFpga_cRIODAQDMA_9159_IndicatorArrayU8;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_IndicatorArrayU8Size_DMATtoHOSTFrameType = 1,
   NiFpga_cRIODAQDMA_9159_IndicatorArrayU8Size_DMATtoHOSTSampleSize = 1,
   NiFpga_cRIODAQDMA_9159_IndicatorArrayU8Size_FPGAVIversion = 2,
} NiFpga_cRIODAQDMA_9159_IndicatorArrayU8Size;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_IndicatorArrayU16_DMATtoHOSTBlockNWords =
0x818E,
   NiFpga_cRIODAQDMA_9159_IndicatorArrayU16_DMATtoHOSTNCh = 0x8176,
   NiFpga_cRIODAQDMA_9159_IndicatorArrayU16_InsertedIOModulesID =
0x81B0,
} NiFpga_cRIODAQDMA_9159_IndicatorArrayU16;
```

```
typedef enum
{
   NiFpga_cRIODAQDMA_9159_IndicatorArrayU16Size_DMATtoHOSTBlockNWords =
1,
   NiFpga_cRIODAQDMA_9159_IndicatorArrayU16Size_DMATtoHOSTNCh = 1,
   NiFpga_cRIODAQDMA_9159_IndicatorArrayU16Size_InsertedIOModulesID =
16,
} NiFpga_cRIODAQDMA_9159_IndicatorArrayU16Size;

typedef enum
{
   NiFpga_cRIODAQDMA_9159_TargetToHostFifoU64_DMATtoHOST0 = 0,
} NiFpga_cRIODAQDMA_9159_TargetToHostFifoU64;

#endif
```

Appendix B. STAND-ALONE CRIO DAQ APPLICATIONS SOURCECODE

cRIO Point by Point DAQ Stand-Alone C Application

```
#include <unistd.h>
#include <sys/time.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <math.h>
#include <errno.h>
#include <irioDriver.h>
#include <irioDataTypes.h>
#include <irioHandlerAnalog.h>
#include <irioHandlerDigital.h>
#include <irioHandlerDMA.h>
#include <irioHandlerImage.h>
#include <irioHandlerSG.h>
#include <irioResourceFinder.h>
#define msgtest(n,function)  \

printf("#####################################################\n"); \
printf("iRIOCore test" #n " " #function "function\n"); \
printf("#####################################################\n")

#define msgerr(a,n,f,stat) if (a==IRIO_success){\
printf("iRIOCore test" #n " " #f " function successful execution [OK]
\n");fflush(stdout);\
                                        }\
else{\
printf("iRIOCore test" #n " " #f " function unsuccessful execution
[ERROR] %s\n",status.msg);fflush(stdout);\

free(stat.msg);stat.msg=NULL;stat.code=IRIO_success;exit(1);\
                                        }

void usage(char *name) {
        printf("This example checks the CRIO I/O FPGA design\n"
                "This example requires a cRIO device\n"
                "Use lsrio.py to identify the RIO devices included in
the fast controller\n"
                        "\n"
                "Usage: ./%s <SERIAL_NUMBER> <cRIO MODEL> \n"
                "Example: %s 01666C59 9159\n", name, name);
}

int main (int argc, char **argv)
{
        irioDrv_t p_DrvPvt;
        TStatus status;
        status.msg=NULL;
```

```
        status.code=IRIO_success;
        TIRIOStatusCode st=IRIO_success;
        int32_t aivalue;
        size_t valueLength;
        char VIversion[4];
        char *currPath=NULL;
        char *bitFilePath=NULL;
        char *bitfileName=NULL;

        char criomodel[20];
        char NIcriomodel[20];
        char criomodel_serial[20];

        if (argc!=3)
        {
                usage(argv[0]);
                return 1;
        }
        if( (strlen(argv[1]) > 20) || (strlen(argv[2])>20) )
        {
                usage(argv[0]);
                return 1;
        }
        strcpy(criomodel_serial,argv[1]);
        strcpy(criomodel,argv[2]);

        bitfileName=(char*)malloc(128);

        sprintf(bitfileName,"cRIOIO_%s",criomodel);
        sprintf(NIcriomodel,"NI %s",criomodel);
        currPath = get_current_dir_name();

        asprintf(&bitFilePath,"%s/resourceTest/%s/",currPath,criomodel);
        free(currPath);
        printf ("Pwd: %s \n", bitFilePath);

        //CHASSIS1 N/S: 19ED079 3x9205, 1x9264, 1x9401 1x9426 2x9477

        msgtest(0,irio_initDriver);

st|=irio_initDriver("test",criomodel_serial,NIcriomodel,bitfileName,"V1.
1",1,bitFilePath,bitFilePath,&p_DrvPvt,&status );
        msgerr(st,0,irio_initDriver,status)


        msgtest(1,irio_setFPGAStart);
        st|=irio_setFPGAStart(&p_DrvPvt,1, &status);
        msgerr(st,1,irio_setFPGAStart,status)

        msgtest(1,irio_getFPGAStart);
        st|=irio_getFPGAStart(&p_DrvPvt, &aivalue,&status);
        printf("FPGA Start Value %d \n", aivalue);
        msgerr(st,1,irio_getFPGAStart,status);

        if (p_DrvPvt.cRIOModulesOK==0) printf("cRIOModules are not
matching expected configuration \n");
```

```
        msgtest(2,irio_setDAQStartStop);
        st|=irio_setDAQStartStop(&p_DrvPvt,1,&status);
        msgerr(st,2,irio_setDAQStartStop,status);

        st|=irio_getDAQStartStop(&p_DrvPvt,&aivalue,&status);
        printf("DAQStartStop Value %d \n", aivalue);
        msgerr(st,2,irio_getDAQStartStop,status);
        msgtest(3,irio_getFPGAVIVersion);

        st|=irio_getFPGAVIVersion(&p_DrvPvt,VIversion, 4,
&valueLength,&status);
        printf("IRIOCore test 3: VI version: %d.%d
\n",VIversion[1],VIversion[3]);
        msgerr(st,3,irio_getFPGAVIVersion,status);

        msgtest(3,irio_getDevQualityStatus);
        st|=irio_getDevQualityStatus(&p_DrvPvt,&aivalue,&status);
        printf("Device Quality Status Value %d \n", aivalue);
        msgerr(st,3,irio_getDevQualityStatus,status);

        st|=irio_getDevTemp(&p_DrvPvt,&aivalue,&status);
        printf("Device Temperature %f \n", aivalue*0.25);
        msgerr(st,3,irio_getDevTemp,status);

        st|=irio_getDevProfifle(&p_DrvPvt,&aivalue,&status);
        printf("Device Profile %d \n", aivalue);
        msgerr(st,3,irio_getDevProfifle,status);

        msgtest(4,Analog Input Acquisition);
        st|=irio_setDebugMode(&p_DrvPvt,0,&status);
        printf("DebugMode Value Set: 0 \n");
        msgerr(st,4,irio_setDebugMode,status);
        usleep(100000);

        st|=irio_getAI(&p_DrvPvt,0,&aivalue,&status);
        printf("Slot 2 9205 AI0 Value %f \n", aivalue*p_DrvPvt.CVADC);
        msgerr(st,4,irio_getAI,status);

        st|=irio_getAI(&p_DrvPvt,1,&aivalue,&status);
        printf("Slot 2 9205 AI1 Value %f \n", aivalue*p_DrvPvt.CVADC);
        msgerr(st,4,irio_getAI,status);

        st|=irio_getAI(&p_DrvPvt,2,&aivalue,&status);
        printf("Slot 2 9205 AI2 Value %f \n", aivalue*p_DrvPvt.CVADC);
        msgerr(st,4,irio_getAI,status);

        st|=irio_setDebugMode(&p_DrvPvt,1,&status);
        printf("DebugMode Value Set: 1 \n");
        msgerr(st,4,irio_setDebugMode,status);
        usleep(100000);

        st|=irio_getAI(&p_DrvPvt,0,&aivalue,&status);
        printf("Emulated AI0 Value %f \n", aivalue*p_DrvPvt.CVADC);
        msgerr(st,4,irio_getAI,status);

        st|=irio_getAI(&p_DrvPvt,1,&aivalue,&status);
        printf("Emulated AI1 Value %f \n", aivalue*p_DrvPvt.CVADC);
```

```
        msgerr(st,4,irio_getAI,status);

        st|=irio_getAI(&p_DrvPvt,2,&aivalue,&status);
        printf("Emulated AI2 Value %f \n", aivalue*p_DrvPvt.CVADC);
        msgerr(st,4,irio_getAI,status);

        msgtest(5,Analog Output and Input Acquisition);

        st|=irio_setDebugMode(&p_DrvPvt,0,&status);
        printf("DebugMode Value Set: 0 \n");
        msgerr(st,5,irio_setDebugMode,status);
        usleep(100000);

        st|=irio_setAOEnable(&p_DrvPvt,0,NiFpga_True, &status);
        printf("AOEnable0 Value to Set: Enable \n");
        msgerr(st,5,irio_setAOEnable,status);

        st|=irio_getAOEnable(&p_DrvPvt,0,&aivalue,&status);
        printf("AOEnable0 Value: %d \n",aivalue);
        msgerr(st,5,irio_getAOEnable,status);

        st|=irio_setAO(&p_DrvPvt,0,4000, &status);
        printf("AO0 Value to Set: %f \n",4000/p_DrvPvt.CVDAC);
        msgerr(st,5,irio_setAO,status);

        st|=irio_getAO(&p_DrvPvt,0,&aivalue,&status);
        printf("AO0 Value: %f \n", aivalue/p_DrvPvt.CVDAC);
        msgerr(st,5,irio_getAO,status);
        usleep(100000);

        st|=irio_getAI(&p_DrvPvt,0, &aivalue, &status);
        printf("Slot 2 9205 AI0 Value %f \n", aivalue*p_DrvPvt.CVADC);
        msgerr(st,5,irio_getAI,status);

        st|=irio_setAO(&p_DrvPvt,1,8000, &status);
        printf("AO1 Value to Set: %f \n",8000/p_DrvPvt.CVDAC);
        msgerr(st,5,irio_setAO,status);

        st|=irio_getAO(&p_DrvPvt,1,&aivalue,&status);
        printf("AO1 Value: %f \n", aivalue/p_DrvPvt.CVDAC);
        msgerr(st,5,irio_getAO,status);

        st|=irio_setAOEnable(&p_DrvPvt,1,NiFpga_True, &status);
        printf("AOEnable1 Value to Set: Enable \n");
        msgerr(st,5,irio_setAOEnable,status);

        st|=irio_getAOEnable(&p_DrvPvt,1,&aivalue,&status);
        printf("AOEnable1 Value: %d \n", aivalue);
        msgerr(st,5,irio_getAOEnable,status);
        usleep(100000);

        st|=irio_getAI(&p_DrvPvt,1, &aivalue, &status);
        printf("Slot 2 9205 AI1 Value %f \n", aivalue*p_DrvPvt.CVADC);
        msgerr(st,5,irio_getAI,status);

        st|=irio_setAO(&p_DrvPvt,2,-8000, &status);
        printf("AO2 Value to Set: %f \n", -8000/p_DrvPvt.CVDAC);
```

```
        msgerr(st,5,irio_setA0,status);

        st|=irio_getAO(&p_DrvPvt,2,&aivalue,&status);
        printf("AO2 Value: %f \n", aivalue/p_DrvPvt.CVDAC);
        msgerr(st,5,irio_getAO,status);

        st|=irio_setAOEnable(&p_DrvPvt,2,NiFpga_True, &status);
        printf("AOEnable2 Value to Set: 1 \n");
        msgerr(st,5,irio_setAOEnable,status);

        st|=irio_getAOEnable(&p_DrvPvt,2,&aivalue,&status);
        printf("AOEnable2 Value: %d \n", aivalue);
        msgerr(st,5,irio_getAOEnable,status);

        usleep(100000);
        st|=irio_getAI(&p_DrvPvt,2, &aivalue, &status);
        printf("Slot 2 9205 AI2 Value %f \n", aivalue*p_DrvPvt.CVADC);
        msgerr(st,5,irio_getAI,status);

        msgtest(6,Digital Inputs and Outputs);
        st|=irio_setDO(&p_DrvPvt,0,1, &status);
        printf("DO0 Value to Set: 1 \n");
        msgerr(st,6,irio_setDO,status);

        st|=irio_getDO(&p_DrvPvt,0,&aivalue, &status);
        printf("DO0 Value: %d \n", aivalue);
        msgerr(st,6,irio_getDO,status);
        usleep(100000);

        st|=irio_getDI(&p_DrvPvt,0,&aivalue, &status);
        printf("NI 9401 DI Value Read %d \n", aivalue);
        msgerr(st,6,irio_getDI,status);

        st|=irio_setDO(&p_DrvPvt,0,0, &status);
        printf("DO0 Value to Set: 0 \n");
        msgerr(st,6,irio_setDO,status);

        st|=irio_getDO(&p_DrvPvt,0,&aivalue, &status);
        printf("DO0 Value: %d \n", aivalue);
        msgerr(st,6,irio_getDO,status);
        usleep(1000000);

        st|=irio_getDI(&p_DrvPvt,0,&aivalue, &status);
        printf("NI 9401 DI Value Read %d \n", aivalue);
        msgerr(st,6,irio_getDI,status);

        st|=irio_setDO(&p_DrvPvt,1,1, &status);
        printf("DO1 Value to Set: 1\n");
        msgerr(st,6,irio_setDO,status);

        st|=irio_getDO(&p_DrvPvt,1,&aivalue, &status);
        printf("DO1 Value: %d \n", aivalue);
        msgerr(st,6,irio_getDO,status);
        usleep(1000000);

        st|=irio_getDI(&p_DrvPvt,1,&aivalue, &status);
```

```
        printf("NI9426 DI Generated by NI9477 Value Read %d \n",
aivalue);
        msgerr(st,6,irio_getDI,status);

        st|=irio_setDO(&p_DrvPvt,1,0, &status);
        printf("DO1 Value to Set: 0\n");
        msgerr(st,6,irio_setDO,status);


        st|=irio_getDO(&p_DrvPvt,1,&aivalue, &status);
        printf("DO1 Value: %d \n", aivalue);
        msgerr(st,6,irio_getDO,status);
        usleep(1000000);

        st|=irio_getDI(&p_DrvPvt,1,&aivalue, &status);
        printf("NI9426 DI Generated by NI9477 Value Read %d \n",
aivalue);
        msgerr(st,6,irio_getDI,status);

        st|=irio_setDO(&p_DrvPvt,2,1, &status);
        printf("DO2 Value to Set: 1\n");
        msgerr(st,6,irio_setDO,status);
        usleep(1000000);

        st|=irio_getDI(&p_DrvPvt,2,&aivalue, &status);
        printf("NI425 DI Generated by NI9476 Value Read %d \n",
aivalue);
        msgerr(st,6,irio_getAI,status);


        st|=irio_setDO(&p_DrvPvt,2,0, &status);
        printf("DO2 Value to Set: 0\n");
        msgerr(st,6,irio_getAI,status);
        usleep(1000000);

        st|=irio_getDI(&p_DrvPvt,2,&aivalue, &status);
        printf("NI425 DI Generated by NI9476 Value Read %d \n",
aivalue);
        msgerr(st,6,irio_getDI,status);

        msgtest(7,Auxiliary Inputs and Outputs);
        st|=irio_setAuxAO(&p_DrvPvt,0,5000, &status);
        printf("AuxAO0 Value to Set: 5000\n");
        msgerr(st,7,irio_setAuxAO,status);

        st|=irio_getAuxAO(&p_DrvPvt,0,&aivalue,&status);
        printf("AuxAO0 Value: %d \n", aivalue);
        msgerr(st,7,irio_getAuxAO,status);
        usleep(100000);

        st|=irio_getAuxAI(&p_DrvPvt,0,&aivalue,&status);
        printf("AuxAI0 Value %d \n", aivalue);
        msgerr(st,7,irio_getAuxAI,status);

        st|=irio_setAuxAO(&p_DrvPvt,1,250, &status);
        printf("AuxAO1 Value to Set: 250\n");
        msgerr(st,7,irio_setAuxAO,status);
```

```
        st|=irio_getAuxAO(&p_DrvPvt,1,&aivalue,&status);
        printf("AuxAO1 Value: %d \n", aivalue);
        msgerr(st,7,irio_getAuxAO,status);
        usleep(100000);

        st|=irio_getAuxAI(&p_DrvPvt,1,&aivalue,&status);
        printf("AuxAI1 Value %d \n", aivalue);
        msgerr(st,7,irio_getAuxAI,status);

        st|=irio_setAuxDO(&p_DrvPvt,0,1, &status);
        printf("AuxDO0 Value to Set: 1\n");
        msgerr(st,7,irio_setAuxDO,status);

        st|=irio_getAuxDO(&p_DrvPvt,0,&aivalue,&status);
        printf("AuxDO0 Value: %d \n", aivalue);
        msgerr(st,7,irio_getAuxDO,status);
        usleep(100000);

        st|=irio_getAuxDI(&p_DrvPvt,0,&aivalue,&status);
        printf("AuxDI0 Value %d \n", aivalue);
        msgerr(st,7,irio_getAuxDI,status);

        st|=irio_setAuxDO(&p_DrvPvt,1,1, &status);
        printf("AuxDO1 Value to Set: 1\n");
        msgerr(st,7,irio_setAuxDO,status);

        st|=irio_getAuxDO(&p_DrvPvt,1,&aivalue,&status);
        printf("AuxDO1 Value: %d \n", aivalue);
        msgerr(st,7,irio_getAuxDO,status);
        usleep(100000);

        st|=irio_getAuxDI(&p_DrvPvt,1,&aivalue,&status);
        printf("AuxDI1 Value %d \n", aivalue);
        msgerr(st,7,irio_getAuxDI,status);

        msgtest(8,Stop Acquisition and Close Driver);

        st|=irio_setDAQStartStop(&p_DrvPvt,0,&status);
        printf("DAQStartStop Value to Set: 0\n");
        msgerr(st,8,irio_setDAQStartStop,status);

        st|=irio_closeDriver(&p_DrvPvt, &status);
        msgerr(st,8,irio_closeDriver,status);

        return 0;
}
```

DMA-based cRIO DAQ Stand-Alone C Application

```
#include <unistd.h>
#include <sys/time.h>
#include <time.h>
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <math.h>
#include <errno.h>

#include <irioDriver.h>
#include <irioDataTypes.h>
#include <irioHandlerAnalog.h>
#include <irioHandlerDigital.h>
#include <irioHandlerDMA.h>
#include <irioHandlerImage.h>
#include <irioHandlerSG.h>
#include <irioResourceFinder.h>

#define msgtest(n,function)  \

printf("#####################################################\n"); \
printf("iRIOCore test" #n " " #function "function\n"); \
printf("#####################################################\n")

#define msgerr(a,n,f,stat) if (a==IRIO_success){\
printf("iRIOCore test" #n " " #f " function successful execution [OK]
\n");fflush(stdout);\
                                          }\
else{\
printf("iRIOCore test" #n " " #f " function unsuccessful execution
[ERROR] %s\n",status.msg);fflush(stdout);\

free(stat.msg);stat.msg=NULL;stat.code=IRIO_success;exit(1);\
                                         }

void usage(char *name) {
        printf("This example checks the use of cRIO DAQ profile\n"
                "This example requires a specific chassis
configuration\n"
                "debug_mode=1-> analog input 0 and 1 will have +5.0 and
-5.0\n""debug_mode=0-> MOD 0 AI0= NI9264 CH0/ MOD 1 AI0 = NI9264 CH1\n"
                "\n"
"Usage: %s <s/n> <cRIO model> <debug_mode> <voltage NI9264 CH0 voltage
value> <voltage NI9264 CH1 voltage value> <samples_to_read> \n"
                "Example: %s 0\n", name, name);
}
int main(int argc, char **argv)
{
        irioDrv_t p_DrvPvt;

        TStatus status;
        status.code=IRIO_success;
        status.msg=NULL;
        TIRIOStatusCode st=IRIO_success;

        char *currPath=NULL;
        char *bitFilePath;
        uint64_t *data;
```

```
        uint64_t *data2;

        char *value;
        value=malloc(20*sizeof(char));
        size_t valueLength;
        int32_t aivalues;
        int elementsRead;
        double conversion=4.76837158203125e-7;

        int32_t *dmaaivalues;

        char criomodel[20];
        char NIcriomodel[20];
        char criomodel_serial[20];

        char *intermediatePath;
        char *bitfileName;

        data=malloc(100000*sizeof(uint64_t));
        data2=malloc(100000*sizeof(uint64_t));
        int blocksize=0;


        if (argc != 7) {
            usage(argv[0]);
            return 1;
        }
        if( (strlen(argv[1]) > 10) || (strlen(argv[2])>10) )
        {
                usage(argv[0]);
                return 1;
        }
        strcpy(criomodel_serial,argv[1]);
        strcpy(criomodel,argv[2]);

        bitFilePath=(char*)malloc(1024);
        intermediatePath=(char*)malloc(512);
        bitfileName=(char*)malloc(128);

        sprintf(intermediatePath,"/resourceTest/%s/",criomodel);
        sprintf(bitfileName,"cRIODAQDMA_%s",criomodel);
        sprintf(NIcriomodel,"NI %s",criomodel);

        currPath = get_current_dir_name();

        asprintf(&bitFilePath,"%s/resourceTest/%s/",currPath,criomodel);
        free(currPath);
        printf ("Pwd: %s \n", bitFilePath);


        //CHASSIS1 N/S: 19ED079 3x9205, 1x9264, 1x9401 1x9426 2x9477
        msgtest(0,irio_initDriver);

st|=irio_initDriver("test",criomodel_serial,NIcriomodel,bitfileName,"V1.
1",1,bitFilePath,bitFilePath,&p_DrvPvt,&status );
        msgerr(st,0,irio_initDriver,status);
```

```
        msgtest(1,irio_setUpDMAsTtoHost);
        st|=irio_setUpDMAsTtoHost(&p_DrvPvt,&status);
        msgerr(st,1,irio_initDriver,status);

        st|=irio_cleanDMATtoHost(&p_DrvPvt,0,data,
sizeof(data),&status);
        msgerr(st,1,irio_cleanDMATtoHost,status);
        free(data);

        msgtest(2,irio_setFPGAStart);
        st|=irio_setFPGAStart(&p_DrvPvt,1,&status);
        printf("FPGA Start Value to Set: 1 \n");
        msgerr(st,2,irio_setFPGAStart,status);

        st|=irio_getFPGAStart(&p_DrvPvt, &aivalues,&status);
        printf("FPGA Start Value %d \n", aivalues);
        msgerr(st,2,irio_getFPGAStart,status);


        msgtest(3,irio_setDMATtoHostSamplingRate);
        st|=irio_setDMATtoHostSamplingRate(&p_DrvPvt,0,1, &status);
        printf("DMATtoHostSamplingRate0 Value to Set:
%dHz\n",p_DrvPvt.Fref);
        msgerr(st,3,irio_getFPGAStart,status);



st|=irio_getDMATtoHostSamplingRate(&p_DrvPvt,0,&aivalues,&status);
        printf("DMATtoHostSamplingRate0 Value %dHz\n",
aivalues*p_DrvPvt.Fref);
        msgerr(st,3,irio_getDMATtoHostSamplingRate,status);
        msgtest(4,common resources);

        st|=irio_getFPGAVIVersion(&p_DrvPvt,value,10,
&valueLength,&status);
        printf("IRIOCore test 4: VI version: %s\n",value);
        msgerr(st,4,irio_getFPGAVIVersion,status);

        free(value);

        st|=irio_getDevQualityStatus(&p_DrvPvt,&aivalues,&status);
        printf("Device Quality Status Value %d \n", aivalues);
        msgerr(st,4,irio_getDevQualityStatus,status);

        st|=irio_getDevTemp(&p_DrvPvt,&aivalues,&status);
        printf("Device Temperature %fºC \n", aivalues*0.25);
        msgerr(st,4,irio_getDevTemp,status);

        st|=irio_getDevProfifle(&p_DrvPvt,&aivalues,&status);
        printf("Device Profile %d \n", aivalues);
        msgerr(st,4,irio_getDevProfifle,status);

        msgtest(5,configure waveform generator);

        st|=irio_setDebugMode(&p_DrvPvt,atoi(argv[3]),&status);
        //starting DAQ process
        msgerr(st,5,irio_setDebugMode,status);
```

```
        st|=irio_getDebugMode(&p_DrvPvt,&aivalues,&status);
        printf("Debug Mode Value: %d \n", aivalues);
        msgerr(st,5,irio_getDebugMode,status);

        //// Using analog input and outputs


st|=irio_setAO(&p_DrvPvt,0,atof(argv[4])*p_DrvPvt.CVDAC,&status);
        msgerr(st,5,irio_setAO,status);

        st|=irio_getAO(&p_DrvPvt,0,&aivalues,&status);
        printf("AO0 Value to Set: %fV\n", aivalues/p_DrvPvt.CVDAC);
        msgerr(st,5,irio_getAO,status);
        usleep(1000000);


st|=irio_setAO(&p_DrvPvt,1,atoi(argv[5])*p_DrvPvt.CVDAC,&status);
        msgerr(st,5,irio_setAO,status);

        st|=irio_getAO(&p_DrvPvt,1,&aivalues,&status);
        printf("AO1 Value to Set: %fV\n", aivalues/p_DrvPvt.CVDAC);
        msgerr(st,5,irio_getAO,status);
        usleep(1000000);

        st|=irio_setDAQStartStop(&p_DrvPvt,1,&status);
        printf("DAQStartStop Value to Set: 1 \n");
        msgerr(st,5,irio_setDAQStartStop,status);

        st|=irio_getDAQStartStop(&p_DrvPvt,&aivalues,&status);
        printf("DAQStartStop Value: %d \n", aivalues);
        msgerr(st,5,irio_getDAQStartStop,status);

        //Programming the signal generator 0

        st|=irio_setSGUpdateRate(&p_DrvPvt,0,10,&status);
        printf("SGUpdateRate0 Value to Set:
%dHz\n",p_DrvPvt.SGfref[0]/10);
        msgerr(st,5,irio_setSGUpdateRate,status);


        st|=irio_getSGUpdateRate(&p_DrvPvt,0,&aivalues,&status);
        printf("SGUpdateRate0 Value: %dHz
\n",p_DrvPvt.SGfref[0]/aivalues);
        msgerr(st,5,irio_getSGUpdateRate,status);


        int updaterate=1000;

        int freq=100*(4294967296/updaterate); //100 Hz sampled with 1kHz
means 10 samples per cycle
        st|=irio_setSGFreq(&p_DrvPvt,0,freq,&status);
        printf("SGSFreq Value to Set: 100Hz \n");
        msgerr(st,5,irio_setSGFreq,status);

        st|=irio_getSGFreq(&p_DrvPvt,0,&aivalues,&status);
        //aivalues=aivalues*updaterate/4294967296;
```

```
        printf("SGFreq terminal Value: %d \n", aivalues);
        msgerr(st,5,irio_getSGFreq,status);

        st|=irio_setSGSignalType(&p_DrvPvt,0,2,&status); //0=DC, 1=SINE
2=square 3=triangular
        printf("SGSignalType to Set: 2 (Square) \n");
        msgerr(st,5,irio_setSGSignalType,status);

        st|=irio_getSGSignalType(&p_DrvPvt,0,&aivalues,&status);
        printf("SGSignalType Value: %d \n", aivalues);
        msgerr(st,5,irio_getSGSignalType,status);

        st|=irio_setSGAmp(&p_DrvPvt,0,10000,&status);
        printf("SGAmp0 Value to Set: %fV \n",10000/p_DrvPvt.CVDAC);
        msgerr(st,5,irio_setSGAmp,status);

        st|=irio_getSGAmp(&p_DrvPvt,0,&aivalues,&status);
        printf("SGAmp0 Value Set: %fV \n", aivalues/p_DrvPvt.CVDAC);
        msgerr(st,5,irio_getSGAmp,status);

        st|=irio_setSGPhase(&p_DrvPvt,0,0,&status);
        printf("SGPhase0 Value to Set: 0 \n");
        msgerr(st,5,irio_setSGPhase,status);

        st|=irio_getSGPhase(&p_DrvPvt,0,&aivalues,&status);
        printf("SGPhase0 Value Set: %d \n", aivalues);
        msgerr(st,5,irio_getSGPhase,status);

        st|=irio_setAOEnable(&p_DrvPvt,0,1,&status);
        printf("AOEnable0 Value to Set: 1 \n");
        msgerr(st,5,irio_setAOEnable,status);

        st|=irio_getAOEnable(&p_DrvPvt,0,&aivalues,&status);
        printf("AOEnable0 Value: %d \n", aivalues);
        msgerr(st,5,irio_getAOEnable,status);

        st|=irio_setAOEnable(&p_DrvPvt,1,1,&status);
        printf("AOEnable1 Value to Set: 1 \n");
        msgerr(st,5,irio_setAOEnable,status);

        st|=irio_getAOEnable(&p_DrvPvt,1,&aivalues,&status);
        printf("AOEnable1 Value: %d \n", aivalues);
        msgerr(st,5,irio_getAOEnable,status);

        usleep(100000);

        st|=irio_getAI(&p_DrvPvt,1, &aivalues, &status);
        printf("9205 AI1 Value %fV \n", aivalues*p_DrvPvt.CVADC);
        msgerr(st,5,irio_getAI,status);

        st|=irio_setDMATtoHostEnable(&p_DrvPvt,0,1,&status);
        printf("DMATtoHostEnable0 Value to Set: 1\n");
        msgerr(st,6,irio_setDMATtoHostEnable,status);

        st|=irio_getDMATtoHostEnable(&p_DrvPvt,0,&aivalues,&status);
        printf("DMATtoHostEnable0 Value %d \n", aivalues);
        msgerr(st,6,irio_getDMATtoHostEnable,status);
```

```
        blocksize=p_DrvPvt.DMATtoHOSTBlockNWords[0];
        int number_of_blocks=atoi(argv[6])/blocksize;

        dmaaivalues=malloc(number_of_blocks*sizeof(int32_t));

        usleep(1000000); //1 second
        printf("Block length used by DMA: %d\n", blocksize);
        printf("Fref=%dHz\n", p_DrvPvt.Fref);
        int k=0;
        do{
                //attempt to read 1 block
                st|=irio_getDMATtoHostData(&p_DrvPvt, 1, 0, data2,
&elementsRead, &status);
                printf("Elements Read =%d\n", elementsRead*blocksize);
                //elementsRead should be a block if available otherwise
0
                dmaaivalues=(int32_t*)data2;
                if (elementsRead==1){
                        int x,t;

t=sizeof(uint64_t)/p_DrvPvt.DMATtoHOSTSampleSize[0];
                        for (x=0;x<blocksize*t; x++)
                        {               printf("channel[%d]
aivalue[%d]=%f,
\n",x%p_DrvPvt.DMATtoHOSTNCh[0],(x/p_DrvPvt.DMATtoHOSTNCh[0]),dmaaivalue
s[x]*conversion);}
                        k++;
                }
                else usleep(100000); //100ms
        } while (k<number_of_blocks);
        msgerr(st,6,irio_getDMATtoHostData,status);
        free(data2);

        msgtest(7,Using aux terminal );
        st|=irio_setAuxAO(&p_DrvPvt,0,5000, &status);
        printf("AuxAO0 Value to Set: 5000\n");
        msgerr(st,7,irio_setAuxAO,status);

        st|=irio_getAuxAO(&p_DrvPvt,0,&aivalues,&status);
        printf("AuxAO0 Value Set %d \n", aivalues);
        msgerr(st,7,irio_getAuxAO,status);
        usleep(100000);

        st|=irio_getAuxAI(&p_DrvPvt,0,&aivalues,&status);
        printf("AuxAI0 Value %d \n", aivalues);
        msgerr(st,7,irio_getAuxAI,status);

        msgtest(8,Closing DMA and Driver );

        st|=irio_cleanDMAsTtoHost(&p_DrvPvt,&status);
        msgerr(st,8,irio_cleanDMAsTtoHost,status);

        st|=irio_closeDMAsTtoHost(&p_DrvPvt,&status);
        msgerr(st,8,irio_closeDMAsTtoHost,status);

        st|=irio_closeDriver(&p_DrvPvt,&status);
```

```
        msgerr(st,8,irio_closeDriver,status);

        return 0;
}
```